Multi-processeur INRIA:structure et fonctionnement

Jocelyne Erhel, A. Lichnewsky, F. Thomasset

IRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. 954 90 20

Rapports Techniques

N° 14

MULTI-PROCESSEUR-INRIA : STRUCTURE ET FONCTIONNEMENT

Jocelyne ERHEL
Alain LICHNEWSKY
François THOMASSET

Juillet 1982

MULTI-PROCESSEUR-INRIA :
STRUCTURE ET FONCTIONNEMENT

++++++++++++++

Jocelyne ERHEL
Alain LICHNEWSKY
François THOMASSET

+++++++

Abstract :

This report is a technical description of MUPI, a parallel multiprocessor simulated at INRIA on HB-68 (Multics system) ; MUPI has a MIMD architecture, having in view the Cmmp of Carnegie Mellon University. The main physical properties of the machine are parameters of the simulation program : number of processors, processor speed, network performance, memory access time, number of memory banks.

Tools for analysis of results are also presented.

++++++++

Résumé :

Ce manuel décrit le calculateur parallèle, dénommé MUPI, dont l'architecture de type MIMD a été simulée à l'INRIA sur HB-68 (système Multics). La machine qui a servi de modèle est le Cmmp de l'Université de Carnegie Mellon. Les caractéristiques physiques de la machine telles que : nombre de processeurs, temps des instructions, cycle-mémoire, vitesse de traversée du réseau, nombre de bancs-mémoire, sont des paramètres de simulation.

Des outils d'analyse des résultats sont également présentés.

MULTI-PROCESSEUR-INRIA :

STRUCTURE ET FONCTIONNEMENT

Jocelyne ERHEL

Alain LICHNEWSKY

Francois THOMASSET

Juin 1982

--- TABLES DES MATIERES ---

INTRODUCTION.

CHAPITRE 1 : DESCRIPTION GENERALE DE MUPI.
- I. Structure globale de MUPI.
  - 1) Organisation physique.
  - 2) Organisation logique.
- II. Fonctionnement de MUPI.
  - 1) Chargement des programmes en mémoire.
  - 2) Démarrage de la machine.
  - 3) Appel d'un sous-programme.
  - 4) Exécution.
- III. Allocation dans les bancs-mémoire.
  - 1) Allocation des programmes.
  - 2) Allocation des variables.
  - 3) Stockage horizontal.
  - 4) Stockage vertical
  - 5) Stockage cache.
  - 6) Gestion de la place mémoire.

CHAPITRE 2 : DESCRIPTION D'UN PROCESSEUR.
- I. Structure d'un processeur.
  - 1) Instruction
  - 2) Etat d'un processeur.
  - 3) Compteur ordinal.
  - 4) Code condition (CC)
  - 5) Registres de calcul
- II. Pile des programmes.
  - 1) La pile.
  - 2) Haut de pile.
  - 3) Autres éléments de la pile.
- III. Jeu d'instructions d'un processeur.
  - 1) Opérandes des instructions :
  - 2) Types des données.
  - 3) Dimensions des variables et des constantes.
  - 4) Classes d'instructions.
  - 5) Instructions d'arithmétique entière.
  - 6) Instructions d'arithmétique flottante.
  - 7) Instructions de logique et décalages.
  - 8) Instructions de contrôle ordinaire.
  - 9) Instructions de contrôle spécial.
  - 10) Instructions de mesures.
  - 11) Exemples simples de synchronisation.

CHAPITRE 3 : ORGANISATION DU SYSTEME.
- I. Structure interne d'un processeur.
  - 1) Sous-cycles.
  - 2) Structure interne d'un processeur.
  - 3) Situation présente.
  - 4) Mode de communication.

5) Numéro du sous-cycle en cours.
6) Sous-cycles visibles du réseau.
7) Sous-cycles visibles du BIP.

II. Structure du réseau.
1) Caractéristiques.
2) Connexions processeurs-bancs.
3) Conflits d'accès.
4) Durée des connexions.
5) Organisation physique du réseau.
6) Réseau cross-bar.

III. Structure d'un banc-mémoire.
1) Etat d'un banc.
2) Fonctions d'un banc

IV. Communications.
1) Communications processeurs-réseau.
2) Communications processeur-BIP
3) Communications réseau-mémoire.
4) Schéma d'une requête ordinaire.
5) Schéma d'une requête privilégiée.

CHAPITRE 4 : ASSEMBLEUR : STRUCTURE ET SYNTAXE.

I. Structure d'un programme.
1) Description.
2) Déclaration du programme.
3) Déclaration des paramètres.
4) Déclaration des variables locales.
5) Types des données.
6) L'option d'initialisation.
7) Déclaration des constantes numériques.
8) Déclaration des textes de messages.
9)Corps du programme.
10) Fin du programme.
11) Commentaires.
12) Pseudo-instruction PAUSE.
13) Option de trace des tests sur variables réelles.

II. Appel d'un sous-programme.
1) Instructions CALL et ARG
2) Transmissions des arguments.
3) Allocation des variables locales.
4) Sauvegarde du programme appelant.
5) Environnement du programme appelé.

III. Retour d'un sous-programme.
1) Instruction RETURN.
2) Libération des variables locales.
3) Retour du programme appelant.

IV. Cas du programme principal.
1) Environnement du programme.
2) Retour du programme

V. Syntaxe de l'assembleur.
1) Règles générales.
2) Syntaxe des déclarations de programme.
3) Syntaxe des déclarations de paramètres.
4) Syntaxe des déclarations de variables locales.
5) Syntaxe des déclarations de constantes.

6) Syntaxe des déclarations de messages.
7) Syntaxe des instructions éxécutables.
8) Syntaxe des instructions de mesure.

CHAPITRE 5 : OUTILS DE MESURE DANS MUPI.
I. Statistiques.
1) Instruction STAT
2) Statistiques relatives aaux processeurs.
3) Statistiques relatives aux bancs-mémoire.
II. Messages.
III. Etat de la machine.
1) Instruction ETAT.
2) Impression d'un événement.
3) Impression de l'état du réseau.
4) Impression de l'état d'un processeur.
5) Code opération.
6) Problèmes de cohérence.

ANNEXE A : RECAPITULATION DU JEU D'INSTRUCTIONS

ANNEXE B : GESTIONNAIRE DE TABLES DE DONNEES.

ANNEXE C : LOGICIELS D'ANALYSE DES RESULTATS.

ANNEXE D : EXEMPLE D'UTILISATION DE L'ASSEMBLEUR MUPI

ANNEXE E : EXEMPLE DE SIMULATION MUPI

BIBLIOGRAPHIE

Introduction

Le présent rapport a pour objet de décrire le calculateur parallèle dénommé MUPI (Multi-processeur-INRIA), dont l'architecture a été simulée sur l'ordinateur Multics de l'INRIA.

MUPI est un calculateur parallèle de type MIMD (Multiple-Instruction-Multiple-Data), selon la classification de Flynn (réf. 1). La machine qui a servi de modèle est le Cmmp (réf. 2). Les programmes exécutés sur MUPI sont écrits dans le langage assembleur de la machine (chapitre 2 III et chapitre 4).

La modularité du programme de simulation permet de transformer l'un des modules, toutes choses étant égales par ailleurs, et d'étudier l'influence d'un élément de la machine tel que le réseau sur ses performances globales.

Les caractéristiques physiques de la machine telles que nombre de processeurs, vitesse des instructions, cycle-mémoire, vitesse de traversée du réseau sont des paramètres de simulation (cf. liste ci-dessous) que l'on peut faire varier pour mesurer leurs performances relatives.

On peut suivre pas à pas l'éxécution d'un programme en observant l'état de la machine et son évolution au cours du temps. On peut aussi établir des statistiques relatives aux performances d'un programme complet ou d'une séquence déterminée. Ces statistiques mesurent la durée totale d'exécution, la vitesse de chaque processeur, le taux de charge du réseau, le taux d'occupation des bancs-mémoire (cf. chapitre 5).

LISTE DES PARAMETRES DE SIMULATION :

1) Paramètres relatifs à l'architecture :

- Nombre de processeurs P (cf. chap. 1 I)
- Nombre de bancs-mémoire M (cf. chap. 1 I)
- Taille des bancs-mémoire (cf. chap. 1 III)
- Taille des piles des processeurs (cf. chap. 2 II)

2) Durée d'éxécution des instructions
(cf. chap. 2 III)

- Opérations et comparaisons entières
- Opérations et comparaisons flottantes
- Opérations logiques
- Branchements
- Instructions de synchronisation
- Transfers de registres à mémoire
- Appel et retour de sous-programme.

3) Paramètres relatifs au réseau Cross-Bar
(cf. Chap. 3) :

- Temps de traversée (sans mise en attente)
- Temps de mise en file d'attente
- Temps de sortie de file d'attente.

4) Paramètre relatif à la mémoire (cf. Chap. 3) :

- Temps de réponse (lecture/écriture).

CHAPITRE 1 :

DESCRIPTION GENERALE DE MUPI

I. Structure globale de MUPI

II. Fonctionnement de MUPI

III. Allocation dans les bancs-mémoire

I. STRUCTURE GLOBALE :

1) Organisation physique :

MUPI est constitué de :

- P processeurs identiques, numérotés de 1 à P, dotés chacun d'une unité de contrôle et d'une unité de calcul (cf. chapitre 2 et 3.I), qui allouent les variables et exécutent les instructions.
- Une mémoire partagée, divisée en M bancs numérotés de 1 à M, qui contient les instructions et les données (cf. chapitre 1.III, et 3 III).
- Un réseau de communication entre les processeurs et les bancs-mémoires qui assure les transferts des instructions et des données (cf. chap. 3 II et IV).
- Un Bus Inter Processeur, dénommé BIP, par lequel les processeurs envoient des signaux de relance que reçoivent tous les processeurs bloqués dans l'attente précisément d'un tel signal (et qu'ignorent par contre les autres processeurs).

Dans l'attente d'un signal, un processeur stoppe son exécution qui ne reprend qu'après réception du signal sur le BIP. (cf. chap. 3 IV, cf fig.1).

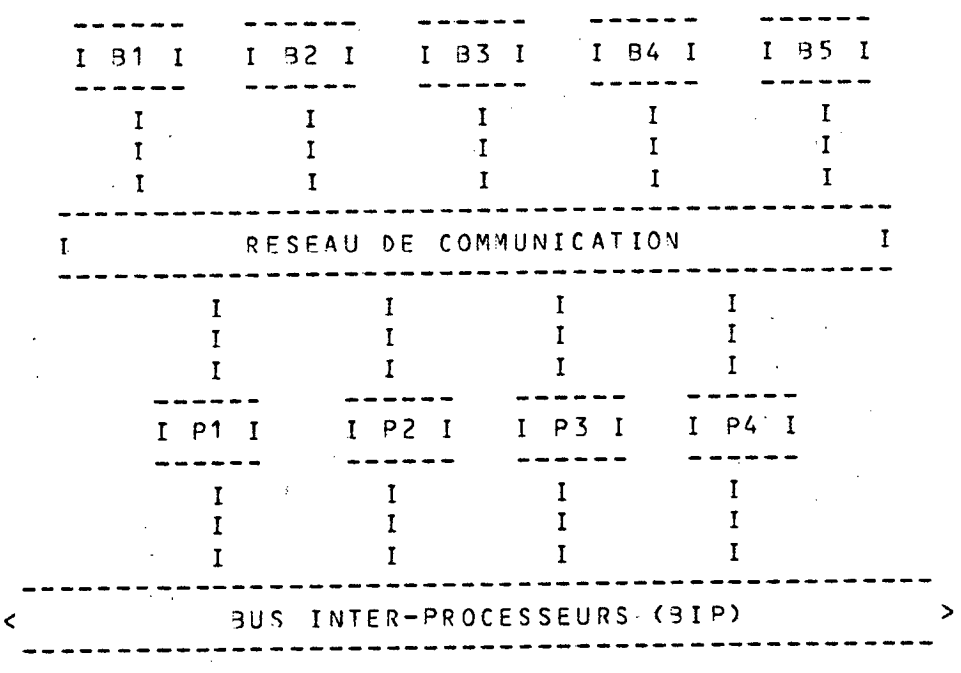


figure 1

ARCHITECTURE GENERALE DE MUPI

2) Organisation logique

Le programme principal, commun à tous les processeurs, répartit les tâches entre eux à l'aide de l'instruction PRN (cf. jeu d'instruction chapitre) qui détermine le numéro du processeur de travail. Que les tâches attribuées aux processeurs soient identiques ou non, leurs exécutions s'effectuent de manière asynchrone. Toute synchronisation entre les tâches doit donc être explicite. Elle peut être réalisée à l'aide des "outils" suivants:

- Un type de variable appelé sémaphore.
- Un mode de communication processeur-banc "privilégié" réalisé par le réseau, permettant de garantir la compatibilité des mises à jour des variables sémaphores.
- Un mécanisme de blocage/relance réalisé sur le BIP, de facon à éviter le recours aux "spin-locks" (attente avec bouclage sur sémaphore).
- Un jeu d'instructions adapté.

Pour lire ou écrire en mémoire, un processeur envoie une requête au réseau, qui la transmet à la mémoire. Un banc donné ne peut acquitter qu'une requête à la fois : la division de la mémoire en bancs permet toutefois d'acquitter M requêtes simultanées. Le réseau gère les requêtes des processeurs de facon à toutes les acquitter en un temps fini tout en résolvant les conflits d'accès aux bancs.

Les processeurs distinguent eux-mêmes les instructions et les données. Aucune instruction ne peut être modifiée en cours d'exécution. Les processeurs partagent tous les programmes et les variables du programme principal qui sont chargés en mémoire avant l'exécution. Ils allouent les variables des sous-programmes à l'appel de ceux-ci.

Un programme ou un tableau de variables peuvent être soit stockés dans un seul banc : le stockage est alors dit vertical, soit répartis dans tous les bancs : le stockage est alors dit horizontal. (cf. Fig. 2 et chapitre 1.II).

L'adressage des variables est indirect. Chaque programme en cours d'exécution possède une table des symboles qui contient les adresses en mémoire de ses variables numérotées et rangées dans l'ordre.
Pour retrouver l'adresse des variables, les instructions font référence à leurs numéros dans la table des symboles du programme (cf. Fig. 3 et 4).

B1	B2	B3	B4	B5
x1		y1		
x2		y2		
x3				
x4				

STOCKAGE VERTICAL :
-de x dans le banc B1
-de y dans le banc B3

B1	B2	B3	B4	B5
x1	x2	x3	x4	x5
x6	x7			
y1	y2	y3	y4	y5

STOCKAGE HORIZONTAL :
-du vecteur x de longueur 7
-du vecteur y de longueur 5

figure 2

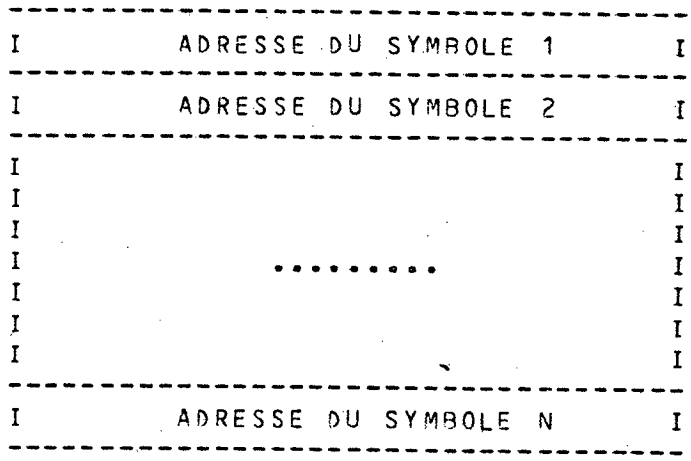
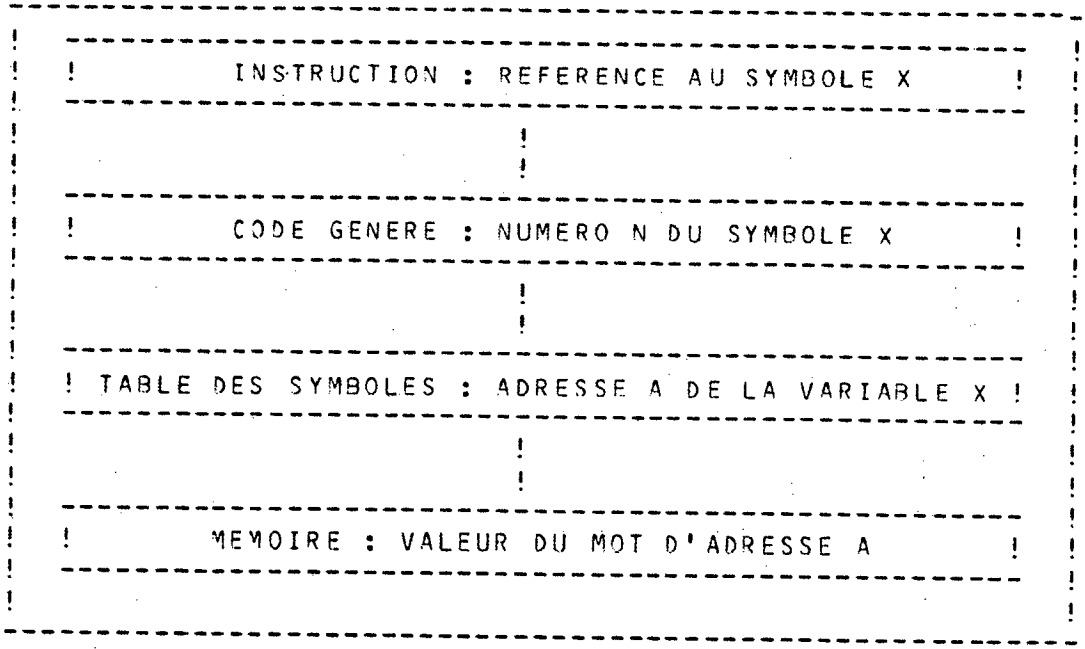


TABLE DES SYMBOLES D'UN PROGRAMME
figure 3



ADRESSAGE INDIRECT DES VARIABLES
figure 4

II. FONCTIONNEMENT DE MUPI

1) Chargement des programmes en mémoire

Les programmes sont écrits dans le langage assembleur de la machine décrit au chapitre 4

Pour chaque programme, l'assembleur génère un code instruction qui est chargé dans les bancs-mémoire avant le démarrage de la machine. Le mode de stockage stipulé dans les déclarations des programmes conditionne l'allocation dans les bancs (cf. III). Eventuellement, des copies du code instruction peuvent être chargées dans des bancs différents.

2) Démarrage de la machine

Au démarrage, l'état de la machine est le suivant :

- aucune transaction n'a lieu entre processeurs et mémoire;
- Pour chaque processeur, les registres, le Code-condition, sont mis à zero;
- Toutes les variables déclarées dans le programme principal ne sont allouées qu'une fois en mémoire. Leurs adresses sont rangées dans la table des symboles associées au programme principal. Chaque processeur contient une copie de cette table des symboles, et partage donc avec les autres processeurs les variables du programme principal (cf Fig. 5).

La première action de chaque processeur est d'éxécuter la première instruction du programme principal; la suite des instructions éxécutées peut cependant dépendre du numéro de processeur, que l'instruction PRN permet de déterminer (cf. jeu d'instructions chap. 2 III).

3) Appel d'un sous-programme

L'instruction CALL réalise l'appel d'un sous-programme (cf. jeu d'instructions chapitre 2 III). La récursion est autorisée.

A chaque appel d'un même sous-programme, le contexte d'appel est sauvegardé dans la pile associée au processeur éxécutant le CALL; les variables locales du sous-programme sont allouées dans les bancs mémoire, et leurs adresses sont rangées dans la table des symboles associée à cet appel.

A un instant donné, un même sous-programme peut donc avoir plusieurs tables des symboles correspondant à des contextes différants (contenant des adresses différentes), que ce soit lié
à un appel récursif ou à une exécution simultanée sur plusieurs processeurs (cf. fig. 6).

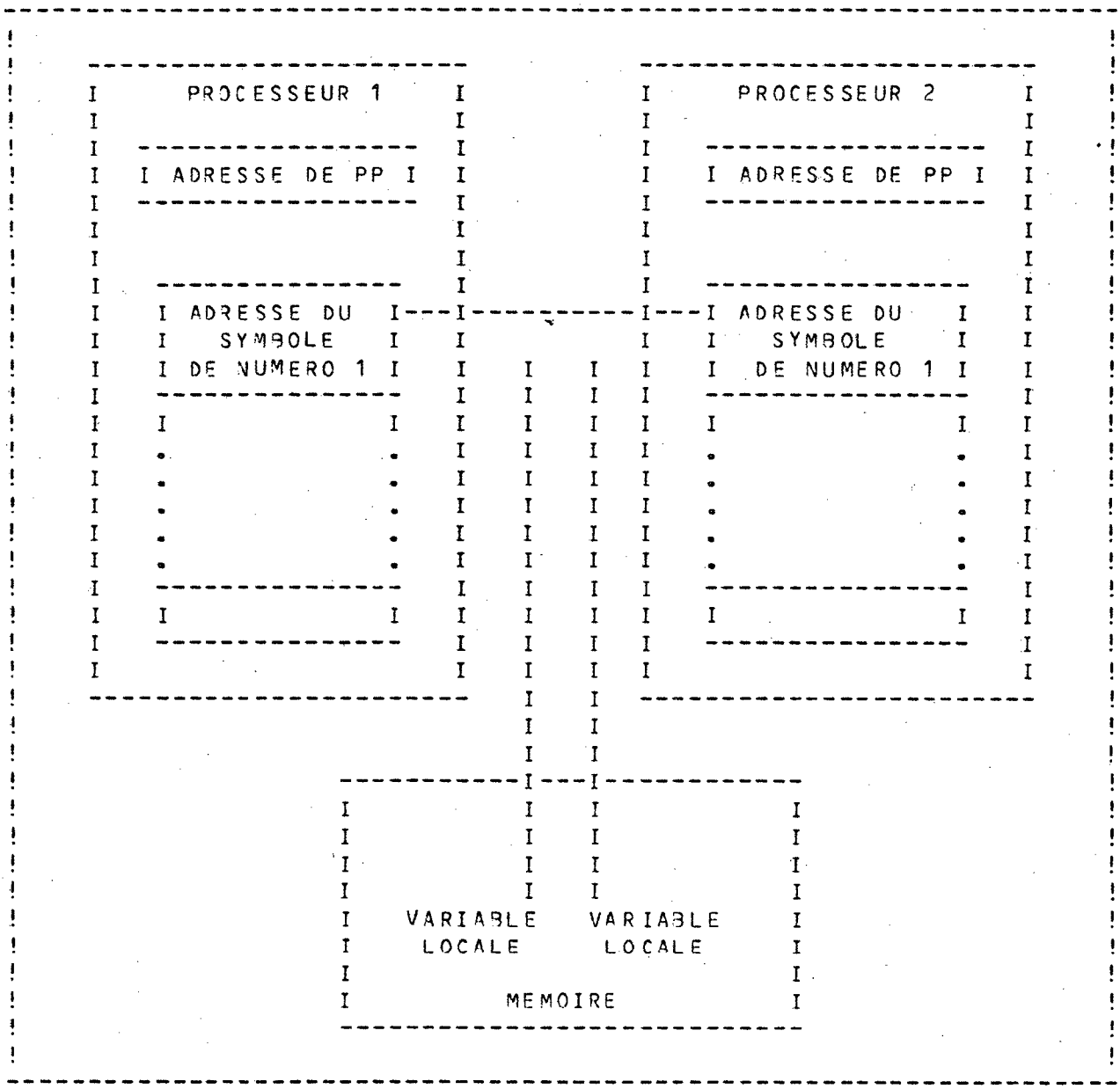
La seule méthode pour partager entre les processeurs des variables d'un sous-programme consiste à les déclarer dans le programme principal et à les transmettre en arguments. Sinon, toutes les variables d'un sous-programme sont locales à chaque appel de ce sous-programme.

4) Exécution

Les processeurs commencent en même temps l'exécution de la 1ère instruction du programme principal, indépendamment les uns des autres et sans autre synchronisation qu'explicite.

Les processeurs sont ininterruptibles ; ils peuvent toutefois mettre en attente d'un signal de relance émis par un autre processeur par l'intermédiaire de BIP.

La 1ère instruction RETURN du programme principal termine l'éxécution de la tâche affectée au processeur. La machine s'arrête lorsque tous les processeurs ont fini d'éxécuter leur tâche. (cf Fig. 5).



TABLES DES SYMBOLES D'UN SOUS-PROGRAMME
(EXECUTION SIMULTANEE SUR 2 PROCESSEURS)
SP : Sous-Programme
figure 6

III. ALLOCATION DANS LES BANCS-MEMOIRE : ADRESSAGE

1) Allocation des programmes

Un programme est alloué dans les bancs suivant l'un des modes de stockage suivants :

- horizontal (réparti entre les bancs)
- vertical (sur un seul banc).
- cache (avec accès immédiat: sans passer par le réseau, sans ralentir l'instruction).

A chaque instruction codée est associée une adresse.

2) Allocation des variables

Les variables sont allouées dans les bancs suivant l'un des modes de stockage suivants :

- horizontal
- vertical

A chaque variable est associée une adresse.
A un vecteur est associée l'adresse de son 1er élément. L'adresse d'un élément quelconque se calcule à l'aide du déplacement.

3) Stockage horizontal

Les blocs (programmes ou vecteurs) sont alloués par rangées horizontales, chaque rangée contenant 1 mot par banc-mémoire.
Les bancs sont numérotés : B1, B2,...BM.
L'allocation commence toujours au banc B1, et s'effectue suivant le schéma suivant :

1ère rangée : B1, B2,...BM
puis
2ème rangée : B1, B2,...BM
puis
Jème rangée : B1, B2,...BM

Soit X un bloc de longueur L.
On lui alloue, à partir de la 1ère rangée libre, n rangées, où n est défini par :

$$(n-1)M < L < nM+1$$

La dernière rangée peut n'être que partiellement allouée.

- A chaque rangée est associé un numéro, à partir de 1.

L'adresse est constituée du :

* mode de stockage H
* numéro du banc mémoire Bm
* numéro de rangée r

L'adresse du bloc x est :

(H, B1, r) ,

où r est le numéro de la 1ère rangée allouée à x .

- Soit xr, x2,...xL les éléments du bloc x

Soit (H,B1,r) l'adresse du bloc x

Soit xi (0 < i < L+1) un élément de x.
Il existe 2 entiers a et m uniques tels que :

$$i = aM + m \text{ et } 0 < m < M+1$$

L'adresse de xi est (H, Bm, a+r).
En particulier, l'adresse de x1 est (H, B1, r).

- Un exemple d'allocation de 2 blocs x et y est schématisé à la figure 7.

- Le numéro N de la 1ère rangée libre est mis à jour après chaque allocation et chaque libération :
  * Allocation de L mots : N <- N + n
  * Libération de L mots : N <- N - n

avec n défini par : $(n-1)M < L < nM+1$

4) Stockage_vertical

Le bloc entier (programme ou vecteur) est alloué dans un unique banc-mémoire.
Chaque banc possède un système d'adressage vertical, à partir de 1.
L'adresse d'un bloc est constituée :
  * du mode de stockage V
  * du numéro du banc Bm (0 < m < M+1)
  * de l'adresse relative à ce banc r

L'adresse du bloc x alloué sur le banc Bm est (V,Bm,r), où r est la 1ère adresse libre dans le banc Bm.

- Soit x1, x2,...,xL les éléments du bloc x.
Soit Bm le banc contenant x.
Soit (V,Bm,r) l'adresse de x.
Alors l'adresse de xi (0 < i < L+1) est :
(V,Bm,r+i-1)

- Un exemple est schématisé à la figure 7.

- La 1ère adresse libre A dans le banc Bm est mise à jour après chaque allocation et libération dans Bm :
  * allocation de L mots : A <- A + L
  * libération de L mots : A <- A - L

5) Stockage_cache

Ce mode de stockage concerne uniquement les programmes, non les variables.
Le programme est accessible par tous les processeurs sans aucune pénalité de temps. Il n'existe aucun conflit d'accès.
Aucune place mémoire n'est allouée dans les bancs.

Ce mode de stockage est compatible avec le caractère inviolable des programmes qu'on ne peut modifier en cours d'exécution.

Il peut être réalisé physiquement en adjoignant une mémoire locale (ou cache) à chaque processeur.

Dans la simulation, la taille d'une mémoire-cache est illimitée.

6) Gestion de la place mémoire

Le stockage horizontal s'effectue par le haut des bancs mémoire.

Le stockage vertical s'effectue par le bas des bancs mémoire.

Les bancs ont tous la même taille T, qui est un paramètre de la machine. La taille d'un banc est le nombre de mots que peut contenir ce banc.

La gestion s'effectue à l'aide des données suivantes :

- H : nombre de mots alloués horizontalement dans chaque banc-mémoire (= nombre de rangées allouées).
- Vm : nombre de mots alloués verticalement dans le banc Bm ($0 < m < M+1$).
- V = max (V1,...Vm).
- T = taille des bancs.

- Le nombre de mots disponibles dans le banc Bm est alors:
  * T - (H + V) pour un stockage horizontal.
  * T - (H + Vm) pour un stockage vertical sur Bm.

A l'allocation du bloc x, l'adresse associée est :
  * H + 1 pour un stockage horizontal
  * Vm + 1 pour un stockage vertical sur Bm.

-cf. fig. 8.

B1	B2	B3	B4	B5	B6	**	NO. DE RANGEE
x1	x2	x3	x4	x5	x6	**	1
x7	x8	x9	x10	x11	x12	**	2
x13	x14	x15	--	--	--	**	3
y1	y2	y3	y4	y5	y6	**	4
y1	y2	y3	y4	$\tilde{y5}$	y6	**	5
--	--	--	--	--	--	**	
--	--	--	--	--	--	**	
--	--	--	--	--	--	**	
t4	--	--	--	--	--	**	4
t3	--	--	w3	--	--	**	3
t2	u2	--	w2	--	--	**	2
t1	u1	--	w1	z1	--	**	1

Figure 7 : Exemple de stockages horizontal et vertical

(x, y : stockage horizontal; t, u, w, z : stockage vertical)

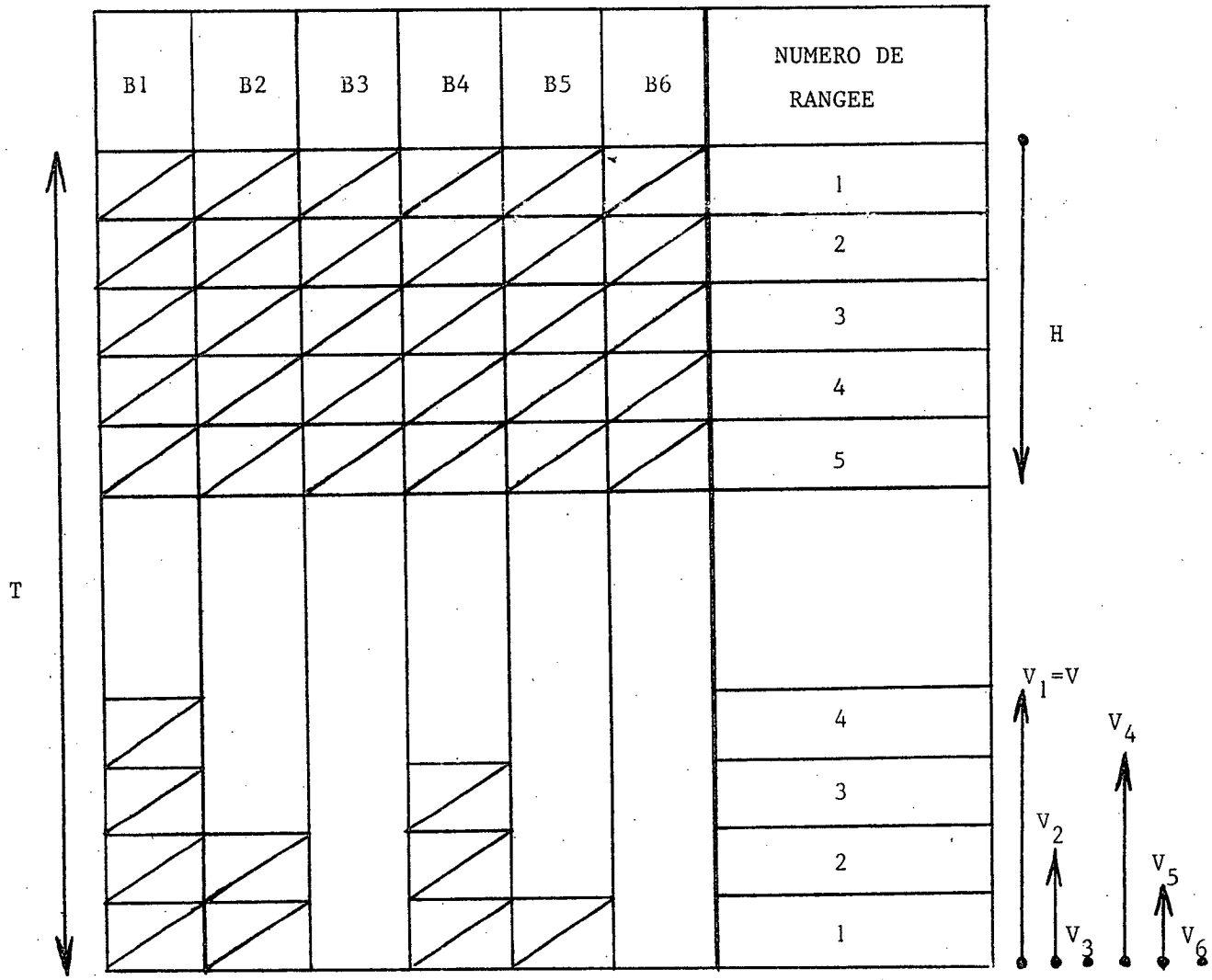


Figure 8

GESTION DE LA PLACE MEMOIRE

CHAPITRE 2 :

DESCRIPTION D'UN PROCESSEUR

I. Structure d'un processeur

II. Pile des programmes

III. Jeu d'instructions

I. STRUCTURE D'UN PROCESSEUR (NIVEAU INSTRUCTION)

1) Instruction

L'instruction est l'unité d'exécution d'un programme. Au cours d'une instruction, l'état du processeur est invisible par le programme qui ne perçoit les changements d'état du processeur qu'entre deux instructions consécutives.

Par contre, au cours d'une instruction, un processeur peut envoyer des requêtes au réseau, des signaux à travers le BIP, ou attendre une relance sur le BIP. Le réseau et le BIP perçoivent donc l'état interne du processeur en train d'exécuter une instruction. Les chapitres suivants décrivent ce niveau interne de structure et l'organisation du système (réseau-processeur-mémoire-BIP) à ce niveau.

REMARQUE : L'utilisateur prendra garde de distinguer les états architecturés (perçus au niveau des instructions), des états intermédiaires, liés à la réalisation des processeurs à un moment donné. Lors de l'évolution du simulateur, la mise d'options diverses conduira à modifier très sensiblement la nature et les séquences des états internes des processeurs.

2) Etat d'un processeur
(tel que accessible au programmeur)

Chaque processeur contient :
- un compteur ordinal : C.O.
- un code condition : C.C.
- 16 registres de calcul numérotés de 0 à 15 : R0, R1,...,R15.
- une pile des programmes.
- identification du sous-programme.
- 2 bits donnant la situation du processeur : libre/attente/bloqué sur BIP/actif.

3) Compteur ordinal

Le compteur ordinal contient l'adrese de l'instruction courante. Cette adresse est relative au début du programme en cours d'exécution.

Le compteur initial est initialisé à 1 au démarrage de la machine et à chaque chargement de programme.

En général, après chaque instruction, le compteur ordinal est incrémenté de 1.

Mais, les instructions de rupture de séquence, par exemple les branchements conditionnels, peuvent modifier le compteur ordinal autrement que par incrémentation de 1.

4) Code condition CC

Le code condition peut prendre 3 valeurs : -1, 0, +1.
Il est initialisé à 0 au démarrage de la machine.
Il est modifié par les instructions de comparaison (ICP, FCP, LCP, LMCP) et par les instructions de synchronisation (TS, WAIT). Par contre, les instructions de branchements ou sauts, d'appel ou retour de sous-programme ne modifient par le C.C. (cf. jeu d'instructions, chap 2 III).
Le C.C. permet d'effectuer des ruptures de séquences conditionnelles.

5) Registres de calcul

Les 16 registres de calcul sont des mots de 36 bits.
Ils sont initialisés à 0.
Leur contenu n'est pas modifié lors de l'appel d'un sous-programme. Il est sauvegardé dans la pile et restitué au retour dans le programme appelant.

II. PILE DES PROGRAMMES

1) La pile

Il existe une pile par processeur.

La pile est constituée :
- d'un haut de pile unique qui contient la référence au programme en cours d'exécution, cf. 2.
- d'un nombre fixe d'éléments identiques qui contiennent chacun la sauvegarde d'un programme, cf. 3.

L'instruction CALL réalise l'appel d'un sous-programme. Elle empile la sauvegarde du programme appelant dans un élément de la pile, range la référence du sous- programme appelé dans le haut de pile, et réinitialise l'état du processeur (cf. III).

L'instruction RETURN réalise le retour d'un sous-programme. Elle retire la dernière sauvegarde de la pile, range la référence au programme correspondant dans le haut de pile, et reconfigure le processeur (cf. III).

Le nombre des éléments de la pile, qui est un paramètre de simulation, détermine le niveau maximal d'empilement ou d'imbrication des sous-programmes. Ce mécanisme autorise la récursion des programmes. Un exemple d'empilement est schématisé à la figure 10.

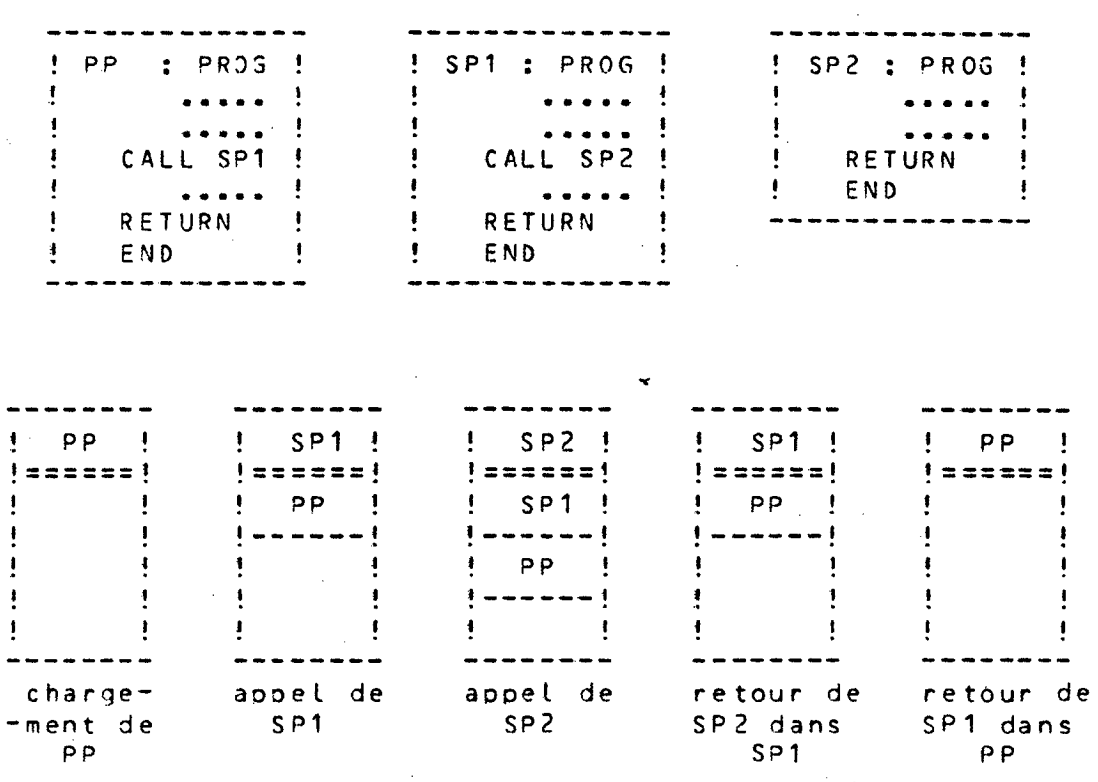


Figure 10 : exemple d'utilisation de la pile

2) Haut_de_pile

Le haut de pile contient la référence au programme en cours d'exécution qui est constituée de :

- l'adresse en mémoire du début de programme (1ère instruction)
- la table des symboles du programme qui contient les adresses de ses variables, générées à l'appel du programme. (cf. III et chapitre 1 II).

L'accès au haut de pile ne pénalise pas la vitesse d'exécution d'un processeur, tout en autorisant un mode de référence symbolique aux variables. Ceci permet en outre la récursion et l'identification automatique des générations de variables.

3) Autres_éléments_de_la_pile

Chaque élément contient la sauvegarde de l'état d'un programme lors d'une instruction CALL, constituée de :

- la référence au programme : adresse en mémoire du début du programme et table des symboles générée à l'appel.
- le compteur ordinal de retour : adresse relative au début du programme de la 1ère instruction exécutable qui suit l'instruction CALL ayant réalisée l'appel du programme.
- Les contenus des registres dans l'état précédant même l'instruction CALL.

III. JEU D'INSTRUCTIONS D'UN PROCESSEUR

1) Opérandes des instructions

Les instructions peuvent faire référence :

- à des variables symboliques contenues dans les bancs-mémoire. L'adressage de ces variables est indirect, via une table des symboles. Le code instruction réfère au numéro de la variable dans la table des symboles qui contient les adresses-mémoire.

- à des constantes symboliques que l'on ne peut ni modifier ni transmettre en arguments et dont les valeurs sont accessibles par le processeur sans perte de temps et sans requête-mémoire.

- à des opérandes immédiats dont la valeur est contenu dans le code instruction.

- aux registres du processeur. Le code instruction contient alors le(s) numéro(s) du ou des registres référencés.

Exemples :

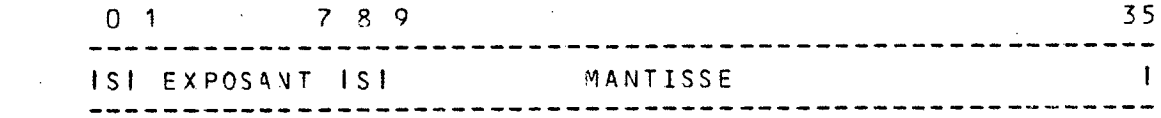
- IADD R0,i ! ajouter au contenu du registre R0 la valeur de la variable symbolique i.
- IADD R0,c ! ajouter au contenu du registre R0 la valeur de la constante symbolique c.
- IADD R0,= 2 ! ajouter 2 au contenu du registre R0
- IADD R0, R1 ! ajouter au contenu du registre R0 le contenu du registre R1.

2) Types des données (variables ou constantes)

Les instructions manipulent trois types de données :

- type entier (IN). Une donnée de type entier est un mot- mémoire de 36 bits.
- Type réel (RE) dans la version actuelle, une option permet de ne pas stocker une donnée de type réel dans le simulateur. Dans ce cas, les instructions d'arithmétique flottante sont simulées mais non exécutées dans Multics. Ce choix permet un gain de place mémoire et de temps calcul dans la simulation.

Quand les variables réelles sont stockées, elles sont conformes au format de MULTICS sur 36 bits :



La valeur numérique flottante correspondante est donnée par :

valeur = 2**E * M;
avec :
E = (-2**7)*bit0 + bit1*2**6 + bit2*2**5 + ... + bit7
M = (-1)*bit8 + bit9*2**(-1) + bit10*2**(-2) + ... + bit36*2**(-27) (Cf. "Multics Processor Manual", AL39, page 2-7)

- Type sémaphore (SM) : Une donnée de type sémaphore est une chaine de 36 bits dont les instructions peuvent marquer chaque bit à "1" b ou "0" b. On peut utiliser un sémaphore comme un verrou constitué d'un jeton et d'une liste d'attente.

Le jeton est le 1er bit (à gauche), qui vaut "1" b si le jeton est pris, "0" b si le jeton est libre.

La liste d'attente est représentée par les 35 autres bits. Le bit (p+1) vaut "1"b si le processeur p est dans la liste d'attente, "0"b sinon. (cf. figure 11). On ne tient pas compte de l'ordre d'arrivée dans la liste d'attente : d'où le terme de "liste" de préférence a "file" d'attente.

Les variables de types entier et réel sont lues ou écrites en mode ordinaire et ne peuvent faire l'objet de requêtes privilégiées.

Par contre, les variables de type sémaphore peuvent être utilisées en mode ordinaire et privilégiée. Leur accès peut donc être protégé, ce qui assure leur cohérence lorsqu'elles sont partagées entre les processeurs. (cf. chapitre 3).

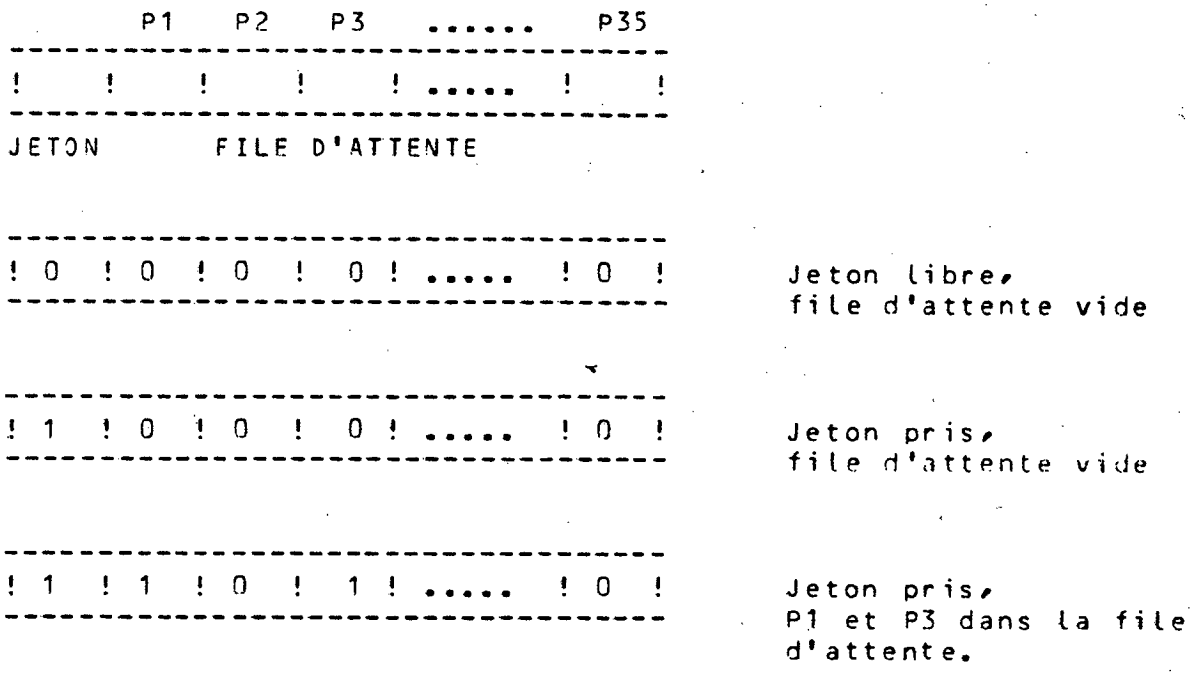


Figure 11 : Variable de type sémaphore :
utilisation en jeton/file d'attente.

3) Dimensions des variables et des constantes :

Les constantes sont toutes des scalaires.
Les variables peuvent être :
- des scalaires : elles sont alors déclarées de longueur 1.
- des vecteurs à 1 dimension : leur déclaration stipule alors leur longueur L, c'est-à-dire leur nombre d'éléments.

L'adresse d'un vecteur est celle de son 1er élément. On retrouve l'adresse d'un élément quelconque à l'aide du déplacement. Le déplacement du 1er élément est nul. Le déplacement correspond à une indexation de 0 à (L-1) des éléments du vecteur.

Scalaires (variables ou constantes) et vecteurs peuvent être de l'un quelconque des trois types entier-réel-sémaphore.

4) Classes d'instructions

Les instructions ont été regroupées en six clases :

- arithmétique entière
- arithmétique flottante
- logiques et décalages
- contrôle ordinaire
- contrôle spécial
- mesures (statistiques, impressions,...).

5) Instructions d'arithmétique entière

Les variables sont déclarées de type entier ou sémaphore.

a) instructions IADD, ISUB, IMUL, IDIV
- 1er opérande :
  dans un registre r
- 2e opérande :
  dans un registre s
  une constante
  immédiat
  une variable de type entier, scalaire
  un élément de tableau de type entier
- résultat dans le registre r du 1er opérande.
- CO incrémenté de 1
- CC inchangé
- IADD : + ; ISUB : - ; IMUL : X ; IDIV : / .

- exemples :

INSTRUCTION	AVANT	APRES
IADD R0,R1	(R0)=1 (R1)=2 CC = 0 CO = 1	(R0)=3 (R1)=2 CC = 0 CO = 2
IMUL R8,=2	(R8)=3 CC = 1 CO = 8	(R8)=6 CC = 1 CO = 9
IDIV R6, i	(R6)=5 i = 2 CC = 1 CO = 3	(R6)=2 CC=-1 CO = 4

b) instruction ICP

- 1er opérande : dans un registre r
- 2e opérande : cf. a)
- pas de résultat
- CO incrémenté de 1
- CC modifié
- effectue la comparaison des 2 opérandes et positionne le CC :

SI	ALORS
op1 < op2	CC <- -1 (<0)
op1 = op2	CC <- 0 (=0)
op1 > op2	CC <- +1 (>0)

- exemples :

INSTRUCTION	AVANT	APRES
ICP R1, R2	(R1)=1 (R2)=2 CC=0 CO=1	(R1)=1 (R2)=2 CC=1 CO=2
ICP R8, nbp	(R8)=2 nbp=2 CC=0 CO=5	(R8)=2 nbp=2 CC=0 CO=6
ICP R6,= 5	(R6)=7 CC=-1 CO=15	(R6)=7 CC=+1 CO=16

c) Instruction ILD

- 1 opérande :
  dans un registre
  une constante
  immédiat
  une variable scalaire de type entier ou TS
  un élément d'un tableau de type entier.
- résultat dans un registre r
- CO incrémenté de 1
- CC inchangé
- charge (Load) dans le regitre résultat r la valeur de l'opérande.

- exemples :

INSTRUCTION	AVANT	APRES
ILD RO,R4	(RO)=1 (R4)=2 CC=0 CO=1	(RO)=2 (R4)=2 CC=0 CC=2
ILD RO,X(3)	(RO)=1 X(3)=5 CC=+1 CO=5	(RO)=5 X(3)=5 CC=+1 CO=6

d) Instruction IST

- 1 opérande dans un registre r
- résultat :
  variable scalaire de type entier ou SM
  élément d'un tableau de type entier
- CO incrémenté de 1
- CC inchangé

- range en mémoire à l'adresse du résultat, la valeur du registre-opérande r (store).

- exemples :

INSTRUCTION	AVANT	APRES
IST R1,i(2)	(R1)=1 i(2)=0 CC=0 CO=3	(R1):1 i(2)=1 CC=0 CO=4
IST R10,N	(R10)=-1 N=2 CC=0 CO=4	(R10)=-1 N=-1 CC=0 CO=5

6) Instructions d'arithmétique flottante

Elles opèrent sur des variables de type réel.
Même instructions qu'en arithmétique entière :

- FADD, FSUB, FMUL, FDIV
- FCP
- FLD
- FST

Remarque : quand l'utilisateur choisit de ne pas simuler les instructions flottantes, le résultat de l'instruction FCP (valeur du CC) est donné par une trace définie avant l'éxécution (Cf. ch 4, I.13)

7) Instructions de logique et décalages

Les instructions opèrent sur des variables de type entier ou sémaphore, considérées comme des chaines de 36 bits.

a) instructions LAND, LOR, LOX
- 1er opérande dans un registre r
- 2ème opérande :
  dans un registre s
  immédiat
  une variable scalaire de type TS ou entier

- résultat dans le registre r du 1er opérande
- CO incrémenté de 1
- CC inchangé
- LAND : "ET" logique ; LOR : "OU" logique ; LOX : "OU EXCLUSIF" logique

- exemples :

INSTRUCTION	AVANT	APRES
LAND R0,R2	(R0)="1100..0"b (R2)="1010..0"b CC=0 CO=1	(R0)="1000..0"b (R2)="1010..0"b CC=0 CO=2
LOR,R2,sema	(R2)="001100..0"b sema="001010..0"b CC=0 CO=2	(R2)="001110..0"b sema="001010..0"b CC=0 CO=3
LOX,R8 arrêt	(R8)= "1100..0"b arrêt="1010..0"b	(R8)= "0110...0"b arrêt="1010..0"b

b) Instruction LNO

- 1 opérande dans un registre r
- résultat dans le registre r de l'opérande
- CO incrémenté de 1
- CC inchangé
- "NOT" logique

- exemples

INSTRUCTION	AVANT	APRES
LNO R11	(R11)="00..0101"b CC=+1 CO=8	(R11)="11..1010"b CC=+1 CO=9
LNO R15	(R15)="011010..01"b CC=-1 CO=5	(R15)="100101..10"b CC=-1 CO=6

c) Instruction LCP

- 1er opérande dans un registre r
- 2e opérande :
  dans un registre s
  immédiat
  variable scalaire de type TS ou entier
- pas de résultat
- CO incrémenté de 1
- CC modifié

- effectue la comparaison des 2 opérandes, et positionne le C.C :

SI	ALORS
op1 = op2	CC=0
op1 ≠ op2	CC=1

- exemples

INSTRUCTION	AVANT	APRES
LCP R12,R13	(R12)="110..0"b (R13)="110..0"b CC=1 CO=3	(R12)="110..0"b (R13)="110..0"b CC=0 CO=4
LCP R11,=5	(R11)="0...1011"b 5 = "0..0101"b CC=0 CO=4	(R11)="0..1011"b 5 = "0..0101"b CC=1 CO=5

- 1er opérande dans un registre r (masque)
- 2e opérande : cf. c)
- pas de résultat
- CO incrémenté de 1
- CC modifié
- effectue la comparaison des deux opérandes, le 1er opérande étant un masque. Le masque est adapté si tout bit valant "1" dans le masque vaut aussi "1" dans le 2e opérande, c'est-à-dire si : (op1 & op2)=op1. Après comparaison, l'instruction positionne le CC.

SI	ALORS
MASQUE ADAPTE	CC=0
MASQUE INADAPTE	CC=1

- exemples

INSTRUCTION	AVANT	APRES
LMCP R0,R1	(R0)="01010..0"b	"
	(R1)="11010..0"b	"
	CC=0 CO=1	CC=0 CO=2
LMCP R0,R2	(R0)="01010..0"b	"
	(R2)="11000..0"b	"
	CC=0 CO=2	CC=1 CO=3

d) Instructions LDD et LDG

- 1er opérande : registre r
- 2e opérande :

immédiat : valeur du décalage p dans un registre s

- résultat : dans le registre r du 1er opérande.
- CO incrémenté de 1
- CC inchangé
- effectue un décalage de p bits dans le registre r.
- LDD : décalage à droite
- LDG : décalage à gauche

- exemples

INSTRUCTION	AVANT	APRES
LDD R0, = 2	(R0)="1010..0"b	(R0)="001010..0"b
	CC=0 CO=1	CC=0 CO=2
LDD R0, = 5	(R0)="0..0100001"b	(R0)="0.....01"b
	CC=0 CO=2	CC=0 CO=3
LDG R1, = 1	(R1)="0..0101"b	(R1)="0..01010"b
	CC=0 CO=3	CC=0 CO=4
LDG R1, R2	(R1)="0101010..0"b	(R1)="1010..0"b
	(R2)= 3	
	CC=1 CO=8	CC=1 CO=9

Remarque : Pour un décalage de p bits à gauche, les bits de gauche (1 à p) sont tronqués. On complète par "0"b à droite. (bits 37-p à 36).

Pour un décalage de p bits à droite, on tronque les bits (37-p à 36) de droite, et on complète par "0"b à gauche bits 1 à p.

8) Instructions de contrôle ordinaire

Ce sont les instructions de branchements et de sauts. Ils peuvent être :

-inconditionnels : le branchement ou saut est toujours effectué

- conditionnels : le branchement ou saut est effectué si le CC est "bien" positionné. Sinon, l'exécution continue en séquence.

a) Branchements

- Ils réfèrent à une étiquette, qui se trouve devant une instruction du programme (cf. assembleur).
- Le CO est modifié en fonction du CC :

Si le CC est "bien" positionné, le CO contient l'adresse relative de l'instruction étiquetée.
Sinon, le CO est incrémenté de 1.

Remarque : Pour les branchements inconditionnels, le CC est toujours "bien" positionné !

- Le CC n'est pas modifié.

b) Sauts

- Ils n'ont aucun opérande.
- Le CO est modifié en fonction du CC :

Si le CC est "bien" positionné, le CO est incrémenté de 2 ; (on saute une instruction).
Sinon, le CO est incrémenté de 1

- Le CC n'est pas modifié.

c) Ruptures de séquence conditionnelles

Il y a rupture si le CO est modifié autrement qu'en l'incrémentant de 1. Sinon, il y a séquence.

CC / CONDITION		-1(<0)	0(=0)	*1(>0)
BL	<0	RUPTURE	SEQUENCE	SEQUENCE
JL	<0			
BE	=0	SEQUENCE	RUPTURE	SEQUENCE
JE	=0			
BG	>0	SEQUENCE	SEQUENCE	RUPTURE
JG	>0			
BLE	< ou = 0	RUPTURE	RUPTURE	SEQUENCE
JLE	< ou = 0			
BNE	< ou > 0	RUPTURE	SEQUENCE	RUPTURE
JNE	< ou > 0			
BGE	> ou = 0	SEQUENCE	RUPTURE	RUPTURE
JGE	> ou = 0			

d) exemples

(debut : CO=1 ; fin : CO=15)

INSTRUCTION	AVANT		APRES	
B debut	CC=0	CO=4	CC=0	CO=1
BL fin	CC=-1	CO=2	CC=-1	CO=15
JGE	CC=-1	CO=4	CC=-1	CO=5
J	CC=0	CO=5	CC=0	CO=7

9) Instructions de contrôle spécial

- instructions de synchronisation : TS, TC, WAIT. Elles opèrent sur des variables de type sémaphore.

Le processeur les exécute en mode privilégié (cf. ch3, I.4).

Durant l'exécution de l'instruction, les autres processeurs n'ont pas accès à la variable concernée.

Le processeur revient en mode courant à la fin de l'instruction

Exemples de TS...TC : cf. figure 12

Exemples de WAIT...TC : cf. figure 13.

- instructions relatives à l'architecture : NB, PRN

Elles permettent de déterminer le nombre de processeurs ou de bancs-mémoire, de repérer le numéro du processeur de travail.

- instructions relatives aux sous-programmes : CALL et RETURN.

a) Instruction TS

- 1 opérande : variable de type sémaphore
- résultat : même variable de type sémaphore
- CO incrémenté de 1 ou inchangé
- CC modifié
- 2 cas d'exécution, en fonction du 1er bit de la variable : (processeur de numéro p).

SI	ALORS	INTERPRETATION : JETON + FILE D'ATTENTE
1e BIT="0"b	-1e BIT = "1"b -CO=CO+1 -CC=+1 - CONTINUER	SI JETON LIBRE ALORS - PRENDRE LE JETON - CONTINUER
1e BIT="1"b	- BIT(p+1)="1"b - CO INCHANGE - CC=0 - ATTENDRE SUR LE - BIP UNE RELANCE D'UN AUTRE PROCESSEUR	SI JETON PRIS ALORS - S'INSCRIRE DANS LA FILE D'ATTENTE - ATTENDRE LA LIBERATION DU JETON

- exemples :

INSTRUCTION	AVANT (P1)	APRES
TS sema	sema="00...0"b CC=0 CO=8	sema="10...0"b CC=1 CO=9 CONTINUE
TS sema	sema="1010...0"b CC=0 CO=5	sema:"1110...0"b CC=0 CO=5 SE BLOQUE SUR BIP

b) Instruction TC

- 1 opérande : variable de type sémaphore
- résultat : même variable de type sémaphore
- CO incrémenté de 1
- CC inchangé
- Le 1er bit doit se trouver à "1"b avant d'exécuter l'instruction, sinon le processeur détecte une erreur et arrête la machine.
- Si le bit (p+1) vaut "1"b, envoyer un signal de relance par le BIP au processeur p.
- Mettre tous les bits du sémaphore à "0"b.
- Interprétation : TC consiste à libérer le jeton et à prévoir tous les processeurs inscrits dans la file d'attente.

- exemples :

INSTRUCTION	AVANT	APRES
TC sema	sema="10...0"b CC=0 CO=5	sema="0...0"b CC=0 CO=6 aucun signal
TC sema	sema="10110...0"b CC=1 CO=8	sema="00...0"b CC=1 CO=9 P2 relancé P2 relancé

c) Instruction WAIT

- 1er opérande : registre (masque = contenu du registre avec 1er bit forcé à "0"b).
- 2e opérande : variable SM
- pas de résultat
- CO incrémenté de 1.
- CC modifié
- 2 cas d'exécution, en fonction du jeton du 2e opérande et du masque.

Le masque est adapté au 2e opérande ssi :
* il est non nul : masque ≠ "0...0"b.
* tout bit valant "1"b dans le masque vaut aussi "1"b dans le 2e opérande : (masque & op2)=masque.

SI	ALORS
JETON LIBRE (1ebit="0"b) OU MASQUE NON ADAPTE	- CC=+1 - CONTINUER
JETON PRIS (1ebit="1"b) ET MASQUE ADAPTE	- CC=0 - SE BLOQUER SUR BIP (ATTENDRE UN SIGNAL

- exemples :

INSTRUCTION	AVANT	APRES
WAIT R0,sema	(R0)="010...0"b sema="11010...0"b CC=-1 C0=2	(R0) = id sema = id CC=0 C0=3 SE BLOQUER SUR BIP
WAIT R0,sema	(R0)="010...0"b sema="01010...0"b CC=0 C0=3	(R0) sema = id CC=+1 C0=4 CONTINUER
WAIT R2,sema	(R2)="110...0"b sema="10110...0"b CC=+1 C0=4	(R2),sema = id CC=+1 C0=5 CONTINUER
WAIT R15,sema	(R15)="110...0"b sema="00...0"b CC= 1 C0=5	(R15),sema = id CC=11 C0=6 CONTINUER

d) Instruction NB

- Pas d'opérande
- Résultat dans un registre r
- CO incrémenté de 1
- CC inchangé
- 1er cas : NB, P

L'instruction charge dans le registre-résultat r la valeur du nombre de processeurs configurés.

- 2e cas : NB, M

L'instruction charge dans le registre-résultat r la valeur du nombre de bancs-mémoire configurés.

- exemples : 3 processeurs et 4 bancs

INSTRUCTION	AVANT	APRES
NB R0, P	(R0)=0	(R0)=3
NB R1, M	(R1)=-1	(R1)=4

d) Instruction PRN

- pas d'opérande
- résultat dans un registre r
- CO incrémenté de 1
- CC inchangé
- 1er cas : PRN

L'instruction charge le bit correspondant au numéro du processeur 1 dans le registre r, et met les autres bits à 0.

Au processeur p est associé le bit (p+1)ème à partir de la gauche.

- 2e cas : PRN, Rv, V

L'instruction charge dans le registre résultat r la valeur p du numéro du processeur de travail.

- exemples (processeur P1) :

INSTRUCTION	AVANT (P1)	APRES (P1)
PRN, R0	(R0)="0...1000"b CC=0 CO=2	(R0)="010...0"b CC=0 CO=3
PRN, R1, V	(R1)=0 CC=0 CO=1	(R1)=1 CC=0 CO=2

e) Instruction CALL

- 1 opérande :
  nom du sous-programme appelé
- CO positionné à 1
- CC inchangé
- contenu des registres inchangé
- appel de sous-programme. La liste des arguments suit l'instruction CALL (cf. instruction ARG). Les arguments peuvent être de tous les types. Ils peuvent être un tableau entier ou un élément de tableau (ils sont alors des scalaires dans le sous-programme). L'ordre de la liste d'arguments doit être le même que dans la liste des paramètres du sous-programme. Les arguments sont transmis par adresse.

L'instruction suivante est la 1ère instruction du sous-programme.

- exemple :

```
PP : PROG                          sp : PROG
x : DEF RE(100),HZ                 x : PAR
y : DEF RE(100),HZ                 a : PAR
CALL sp                            début ...
x : ARG
y : ARG(1)
```

Après le CALL de pp, le processeur exécute la première instruction dans sp (ici étiquetée "début").

Dans sp, x est un tableau de réels de longueur 100, et l'adresse de x(0) est l'adresse de x(0) dans pp; a est un scalaire réel, dont l'adresse est celle de y(1) dans pp.

f) Instruction RETURN

- pas d'opérande
- pas de résultat
- CO positionné à l'adresse de retour dans le programme appelant : c'est l'adresse relative de l'instruction qui suit la liste d'arguments du sous-programme.
- CC inchangé
- registres du programme appelant restaurés.
- retour d'un sous-programme, dans le programme appelant.

La place des variables locales du sous-programme est récupérée.

- exemple

```
PP : PROG                          sp : PROG
 x : DEF RE(100),HZ                x : PAR
 y : DEF RE(100),HZ                a : PAR
       .
       .
 CALL sp                           début : ...
 x : ARG                           ....
 y : ARG(1)                        ....
retour : ...                       ....
         ....                      RETURN
         ...                       END
```

L'instruction exécutée après le RETURN de sp est l'instruction qui suit le CALL dans pp (ici étiquetée "retour").

10) Instructions de mesures

Leur durée d'exécution est nulle. Elles ne génèrent aucune requête mémoire. Ce sont des outils de simulation qui n'ont aucune influence sur les performances de la machine. (cf. chapitre 5).

11) Exemples simples de synchronisation

UN EXEMPLE DE TS ET TC :

Les processeurs P1, P2, P3 éxécutent tous le même programme, et commencent tous en même temps l'instruction 1

```
1 : ...
2 : TS acces
3 : ...
4 : TC acces
5 : ...
```

acces est une variable partagée.
On obtient, par exemple, l'éxécution suivante :

ETAPE	ACCES	ETAT DES PROCESSEURS	COMMENTAIRES
1	0000	n0. * C0 * CC * situation * * * presente P1 * 1 * 0 * actif P2 * 1 * 0 * actif P3 * 1 * 0 * actif	. P1, P2, P3 exécutent la 1ere instruction
2	1000	n0. * C0 * CC * situation * * * presente P1 * 2 * 1 * actif P2 * 2 * 0 * actif P3 * 2 * 0 * actif	. P1 prend le jeton . P2 et P3 demandent le jeton
3	1011	n0. * C0 * CC * situation * * * presente P1 * 3 * 1 * actif P2 * 2 * 0 * bloqué P3 * 2 * 0 * bloqué	. P1 & P2 attendent le jeton; . P1 continue.
4	0000	n0. * C0 * CC * situation * * * presente P1 * 4 * 1 * actif P2 * 2 * 0 * actif P3 * 2 * 0 * actif	. P1 rend le jeton et relance P2 & P3 .P2 et P3 demandent le jeton
5	1001	n0. * C0 * CC * situation * * * presente P1 * 5 * 1 * actif P2 * 3 * 1 * actif P3 * 2 * 0 * bloqué	. P1 continue . P2 prend le jeton et continue . P3 attend le jeton
6	0000	n0. * C0 * CC * situation * * * presente P1 * 6 * 1 * actif P2 * 4 * 1 * actif P3 * 2 * 0 * bloqué	. P1 continue . P2 rend le jeton et relance P3 . P3 demande le jeton

figure 12 (suite page suivante)

7	1000	n0. * CO * CC * situation * * * presente P1 * 7 * 1 * actif P2 * 5 * 1 * actif P3 * 3 * 1 * actif	. P1 continue . P2 continue . P3 prend le jeton
8	1000	n0. * CO * CC * situation * * * presente P1 * 8 * 1 * actif P2 * 6 * 1 * actif P3 * 4 * 1 * actif	. P1 continue . P2 continue . P3 rend le jeton

figure 12 : Un exemple de TS et TC (fin)

UN EXEMPLE DE WAIT ET TC :

Les processeurs P1, P2, P3 exécutent tous le même programme :

```
5 ...
6 PRN R0
7 JNE
8 TC  STOP
9 WAIT R0, STOP
10 ...
```

STOP est une variable partagée (déclarée dans le programme principal).

On obtient, par exemple, l'exécution suivante :

ETAPE	STOP	ETAT DES PROCESEURS	COMMENTAIRES
1	1110	n0.*C0*CC*situation*R0 P1 *6 *1 * actif *0100 P2 *5 *0 * actif *0000 P3 *6 *1 * actif *0001	. P1 & P3 éxécutent l'instruction PRN, leur CC vaut 1 . P2 éxécute l'instruction précédente, son CC=0
2	1110	n0.*C0*CC*situation*R0 P1 *7 *1 * actif *0100 P2 *6 *0 * actif *0010 P3 *7 *1 * actif *0001	. P1 & P3 éxécutent JNE : ils sautent l'instruction : TC STOP . P2 éxécute PRN
3	1110	n0.*C0*CC*situation*R0 P1 *9 *0 * bloqué *0100 P2 *7 *0 * actif *0010 P3 *9 *1 * actif *0001	. P1 éxécute WAIT : il est bloqué car: - jeton de STOP pris - R0 adapté . P2 éxécute WAIT : il continue car : - R0 non adapté . P2 éxécute JNE : il ne saute pas TC
4	0000	n0.*C0*CC*situation*R0 P1 *10*0 * actif *0100 P2 *8 *0 * actif *0010 P3 *10*1 * actif *0001	. P1 est bloqué . P3 continue . P2 éxécute TC STOP : il rend le jeton et relance P1
5	0000	n0.*C0*CC*situation*R0 P1 *10*0 * actif *0100 P2 *9 *1 * actif *0010 P3 *11*1 * actif *0001	. P1 est relancé, il éxécute l'instruction qui suit le WAIT . P2 éxécute WAIT, il continue, car le jeton est libre . P3 continue

figure 13 (suite page suivante)

6	0000	n0.*C0*CC*situation*R0 P1 *11*0 * actif *0100 P2 *10*1 * actif *0010 P3 *11*1 * actif *0010	. P1, P2, P3 continuent

figure 13 : Un exemple de WAIT et TC (fin)

CHAPITRE 3 :

ORGANISATION DU SYSTEME

I. Structure interne d'un processeur

II. Structure du réseau

III. Structure d'un banc-mémoire

IV. Communications :
Requêtes - Blocage/Relance

I. STRUCTURE INTERNE D'UN PROCESSEUR (NIVEAU SOUS-CYCLE)

1) Sous-cycle

Le sous-cycle est la plus petite unité de travail du processeur qui soit visible de l'extérieur (réseau ou BIP). Un sous-cycle est indivisible, reste strictement interne au processeur. Seules les transitions entre deux sous-cycles peuvent être percues de l'extérieur.

Un sous-cycle peut être éxécuté à la suite :

- d'une requête interne du processeur;
- d'une requête du r'\010eseau (réponse à une requête mémoire du processeur);
- d'une requête du BIP (si le processeur était était en attente).

L'éxécution d'un sous-cycle peut provoquer, outre la modification de l'état du processeur :

- une requête au réseau (avec éventuellement mise en attente de la réponse);
- un signal sur le BIP.

Il existe 15 sous-cycles, numérotés de 2 à 16.

On peut classer les sous-cycles en 2 groupes :

- Les sous-cycles qui ne provoquent aucune communication avec l'extérieur ; les transitions entre de tels sous-cycles sont invisibles.
- Les sous-cycles qui peuvent provoquer une communication avec l'extérieur. Les autres éléments de Muppy (réseau, bus et autres processeurs) peuvent alors percevoir la nature de ces sous-cycles.

Ces sous-cycles portent les numéros 4, 5, 7, 12, 14, 15 cf. liste de tous les sous-cycles ci-dessous.

I 2 I	DEBUT DU CYCLE	I
I 3 I	DECODAGE DE L'INSTRUCTION	I
I 4 I	RECHERCHE DU 1er OPERANDE	I
I 5 I	RECHERCHE DU 2e OPERANDE	I
I 6 I	CALCUL	I
I 7 I	STOCKAGE DU RESULTAT	I
I 8 I	COMPTEUR ORDINAL INCREMENTE DE 1	I
I 9 I	RECHERCHE DU DECALAGE	I
I10 I	COMPTEUR ORDINAL MODIFIE EN FONCTION DU CC	I
I11 I	DEBUT DU MODE PRIVILEGIE	I
I12 I	FIN DU MODE PRIVILEGIE	I
I13 I	GESTION DE LA PILE	I
I14 I	BLOCAGE OU NON SELON LE CC	I
I15 I	RELANCE	I
I16 I	FIN DU CYCLE	I

LISTE DES SOUS-CYCLES

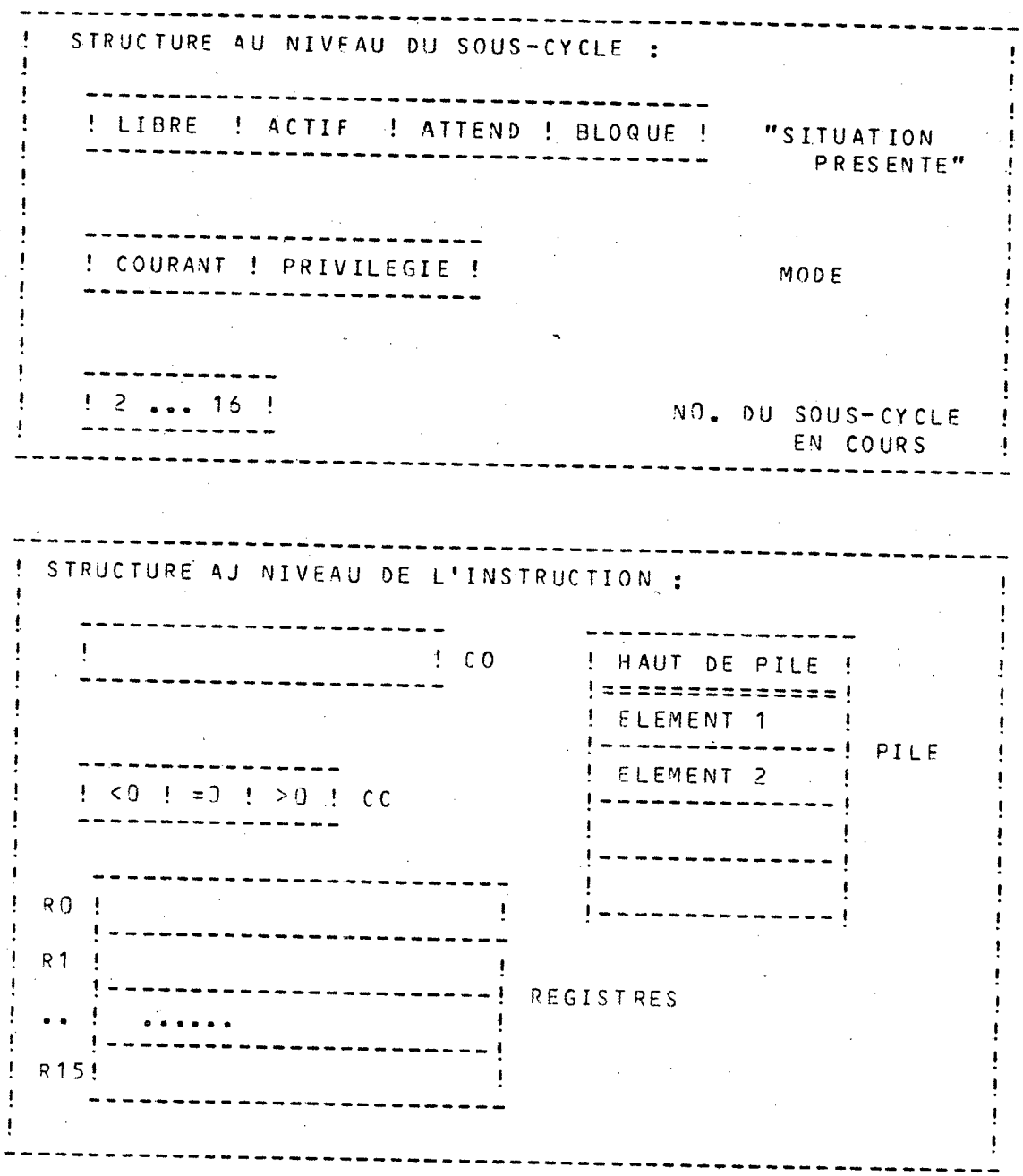


Figure 14 : Structure interne d'un processeur

I. STRUCTURE DU RESEAU

1) Caractéristiques

Le réseau établit les connexions entre les processeurs et les bancs-mémoire. Il assure le transfert des données et des instructions entre les deux. Eventuellement, il gère les conflits d'accès à un banc-mémoire. Toute requête est acquittée au bout d'un temps fini (pas de famine). Les requêtes d'un même processeur sont acquittées dans l'ordre où elles sont reçues.

2) Connexions processeurs-bancs

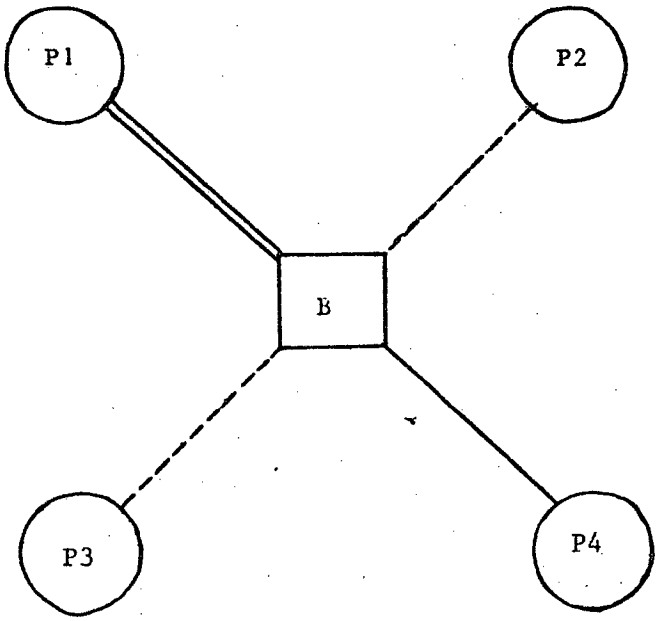
Tous les processeurs peuvent accéder à n'importe lequel des bancs-mémoire.

A un instant donné, un banc-mémoire ne peut être connecté qu'à au plus un processeur. Si deux processeurs veulent accéder en même temps au même banc, il y a un conflit d'accès. La gestion de ces conflits est décrite au paragraphe 3).

De même, à un instant donné, un processeur ne peut accéder qu'à au plus un banc-mémoire. Il obtient aussitot la connexion demandée si le banc est libre, ou est placé dans une file d'attente sinon.

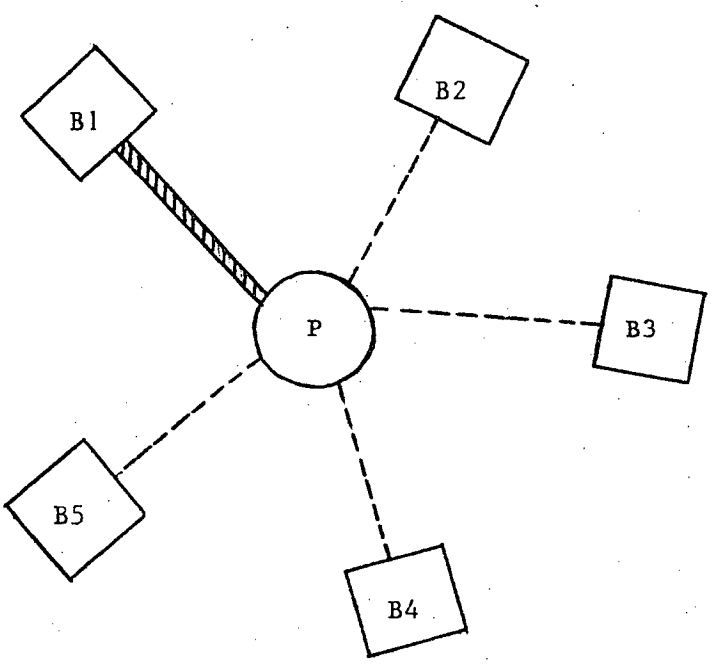
Par contre, tous les bancs peuvent être connectés simultanément, pourvu qu'ils le soient à des processeurs différents. Deux processeurs peuvent donc lire ou écrire en mémoire en même temps, dès qu'ils s'adressent à des bancs différents.

Cette division de la mémoire en bancs minimise les conflits d'accès entre les processeurs.
Les connexions possibles sont schématisées à la figure 15.



Seule la connexion P1-B est établie.

P2, P3, P4 soit ne requièrent pas B, soit sont dans la file d'attente de B.



P est connecté à B1.

Il ne peut être connecté en même temps à B2, B3, B4, B5.

CONNEXIONS PROCESSEURS-BANCS

Figure 15

3) Conflits d'accès

Dès qu'un processeur veut se connecter à un banc déjà occupé, il y a conflit.

Les conflits sont gérés dans des files d'attente, par la règle dite du "PREMIER-ENTRE-PREMIER-SORTI (FIFO). Chaque banc possède sa propre file d'attente.

Lorsqu'un banc est occupé, les requêtes sur ce banc sont placées dans sa file d'attente, dans l'ordre d'arrivée.

Lorsque le banc redevient libre, la première requête de la file d'atente, c'est-à-dire la plus ancienne, peut être acquittée.

Puisque toute connexion dure un temps fini, toutes les requêtes sont satisfaites en un temps fini : par de famine (cf. figures 16 et 17).

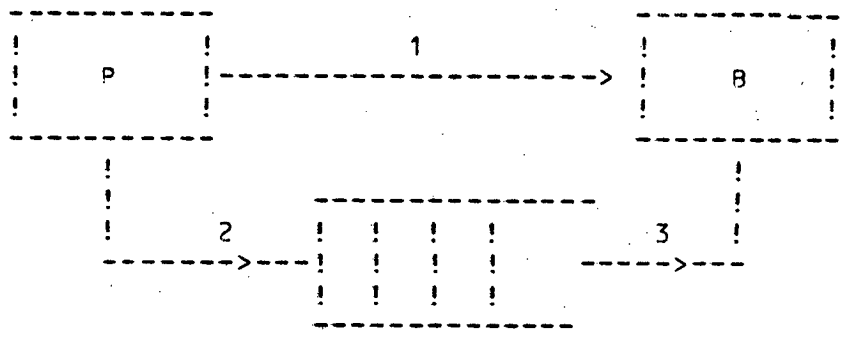


Figure 16 : File d'attente d'un banc

1 : B est libre : P est connecté sans attendre

2 : B est occuppé : P est placé en queue de file d'attente

3 : B est libéré et P est le premier de la file d'attente :

P obtient la connexion avec B

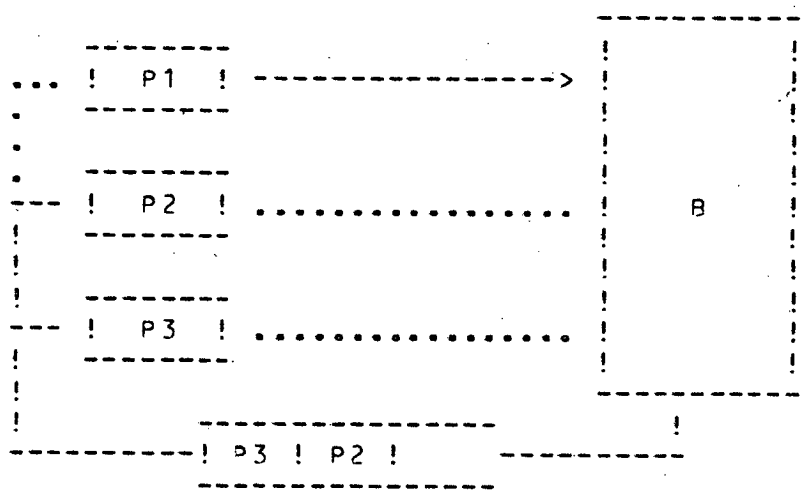


Figure 17 a :

* P1 est connecté a B
* P2 est arrivé le premier sur la file d'attente
* P3 est arrivé après P2 sur la file d'attente

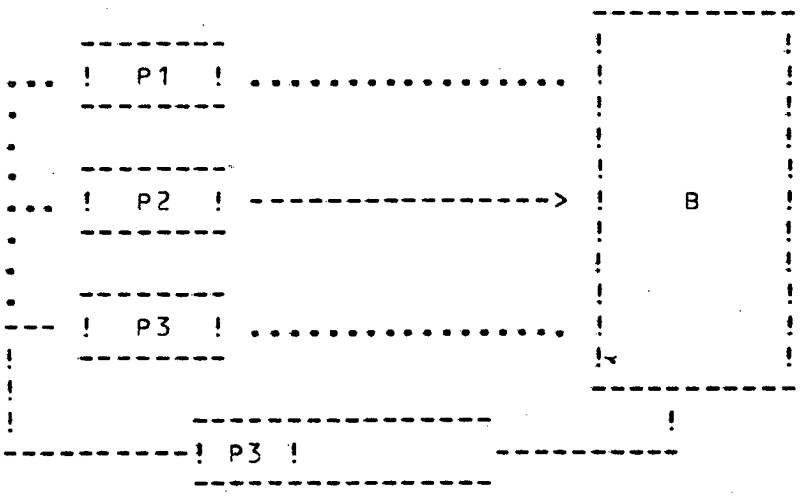


Figure 17 b :
* La connexion P1-B est libérée;
* La connexion P2-B est établie;
* P3 est le premier de la file d'attente.

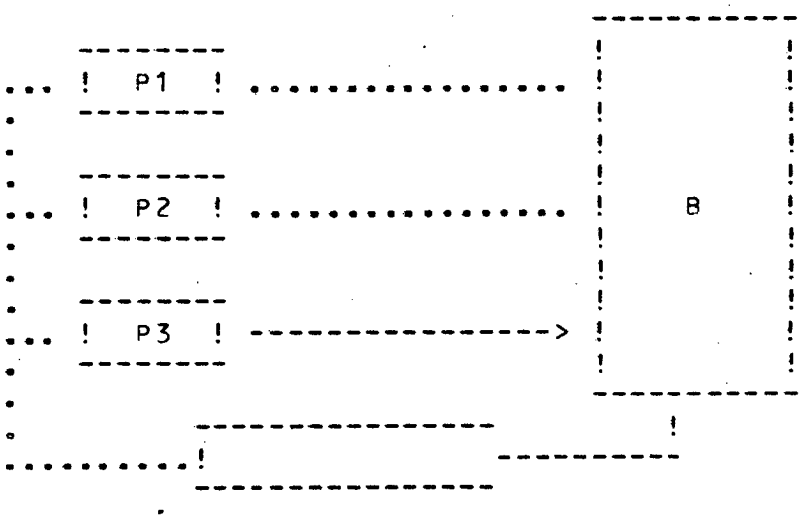


Figure 17 c :
* La connexion P2-B est libérée.
* La connexion P3-B est établie.

Figure 17 : exemple de gestion de file d'attente.

4) Durée_des_connexions

Le réseau établit les connexions :
- soit directement si le banc requis est libre
- soit à la sortie de la file d'attente en cas de conflit.

Les connexions sont de 2 types :

- ordinaire : la connexion est maintenue jusqu'à la réception du signal d'acquittement envoyé par la mémoire. Elle est alors libérée et le réseau envoie un signal d'acquittement au processeur.

- privilégiée : la connexion est maintenue jusqu'à la réception du signal de libération explicite envoyé par le processeur lui-même. Elle est alors libérée.

5) Organisation_physique_du_réseau

Le réseau est un module de la machine, dont l'organisation physique peut être modifiée sans altérer les autres éléments structuraux de la machine. Il suffit de conserver les caractéristiques logiques du réseau énoncées précédemment. Eventuellement, la gestion de la file d'attent et le règlement des conflits peuvent être également changés.

Il est intéressant d'étudier l'influence de l'organisation physique du réseau sur les performances globales de la machine. On pourra étudier, en fonction du réseau, les durées des conflits, les temps d'accès à la mémoire,..., toutes choses étant égales par ailleurs.

Dans la version actuelle de la machine, c'est un réseau de type cross-bar qui est implémenté. Sa structure est décrite au paragraphe suivant.

6) Réseau_cross-bar

Soit P le nombre de processeurs et M le nombre de bancs-mémoire de la machine.
Le réseau contient P x M noeuds, dont l'état représente les connexions éventuelles entre les processeurs et les bancs.
Chaque noeud (p,m), où p est le numéro du processeur et m le numéro du banc-mémoire, peut être :
- libre : aucune connexion entre p et m
- connecté : une connexion ordinaire est établie entre p et m.
- réservé : une connexion privilégiée est établie entre p et m.

Le réseau X-bar est représenté à la figure 18.

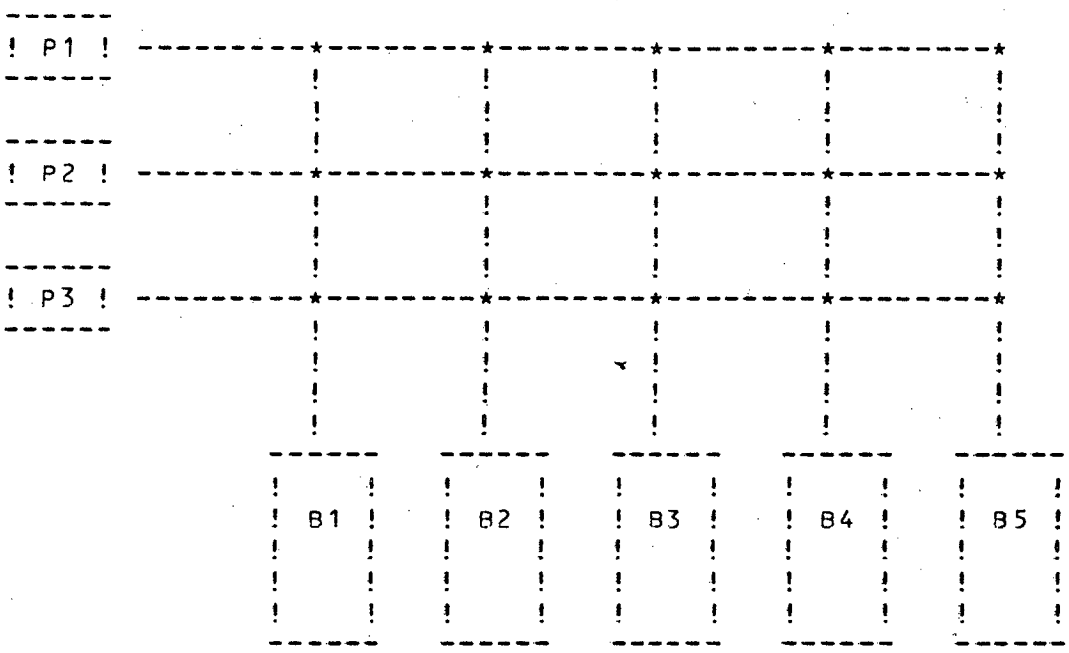


Figure 18 : RESEAU X-BAR

III. STRUCTURE D'UN BANC-MEMOIRE .

1) Etat d'un banc

A un instant donné, un banc mémoire peut être :

- libre : aucun processeur n'a demandé à se connecter sur ce banc.
- occupé : suite à la requête d'un processeur, le banc est connecté.
- réservé : une connexion privilégiée a été établie avec un processeur qui n'a pas encore demandé la libération de cette connexion.

2) Fonctions d'un banc : lectures/ecritures.

Les bancs recoivent deux types de requêtes :

- lecture : ils recoivent l'adresse du mot devant être lu.
- écriture : ils recoivent l'adresse et la valeur du mot à écrire.

Dans les deux cas, ils renvoient un acquittement après avoir exécuté la lecture ou écriture. En cas de lecture, ils transmettent en outre la valeur lue.

Le délai nécessaire à la lecture/écriture d'un mot est un paramètre de la machine.

IV. COMMUNICATIONS ENTRE LES ELEMENTS DE LA MACHINE

1) Communications processeurs - réseau

Toute requête de lecture/écriture dans un banc- mémoire émise par un processeur passe par l'intermédaire du réseau. Inversement, tout acquittement venant de la mémoire passe également par le réseau.

a) Communications processeur - réseau

- émission d'une requête ordinaire de lecture/écriture, contenant l'adresse, et la valeur en cas d'écriture.
- émission d'une requête privilégiée de lecture/écriture, contenant l'adresse, et la valeur en cas d'écriture.
- émission d'une requête de libération de connexion.

Cette demande suit toute requête privilégiée. Elle contient l'adresse concernée par cette requête privilégiée.

b) communications réseau - processeur

- émission d'un acquittement de requête ordinaire de lecture/écriture, contenant la valeur en cas de lecture.

L'acquittement libère en outre la connexion.

- émission d'un acquittement de requête privilégiée de lecture/écriture, contenant la valeur en cas de lecture.

La connexion est réservée jusqu'à libération.

2) Communications processeur - BIP

Le BIP est un Bus Inter-Processeur qui assure les mécanismes de blocage/relance des processeurs.

a) Blocage d'un processeur

Après l'exécution des instructions TS et WAIT (cf. Chap. 2 III) un processeur peut soit continuer l'exécution, soit rester bloqué.

En cas de blocage, un processeur se branche sur le BIP dans l'attente d'un signal de ralance. L'arrivée de ce signal déclenche la fin du blocage, et le processeur peut continuer son exécution.

En même temps qu'il se met à l'écoute du BIP, le processeur bloqué s'inscrit dans la file d'attente correspondante.

b) Relance des processeurs

L'opération TC (cf. chap. 2 III) a pour effet de relancer tous les processeurs se trouvant dans la file d'attente correspondante.

Le processeur qui exécute une instruction TC envoie par le BIP un signal de relance à tous les processeurs en attente dans la file. Ceux-ci sont bloqués à l'écoute du BIP, et sont donc aptes à recevoir le signal qui leur est destiné.

3) Communications réseau - mémoire

Le réseau transmet les requêtes de lecture/écriture à la mémoire et recoit de celle-ci des acquittements.

a) Communications réseau - mémoire

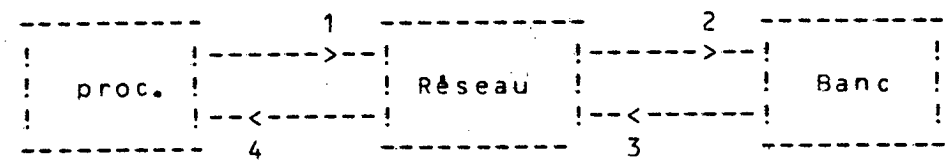
- émission d'une requête ordinaire de lecture/écriture, contenant l'adresse et la valeur en cas d'écriture. Cette émission est précédée de l'établissement de la connexion ordinaire.
- émission d'une requête privilégiée de lecture/écriture, contenant l'adresse et la valeur en cas d'écriture. Cette émission est précédée de l'établissement de la connexion privilégiée.

b) Communications mémoire - réseau

- émission d'un acquittement de requête ordinaire de lecture/écriture, contenant la valeur en cas de lecture.
- émission d'un acquittement de requête privilégiée de lecture/écriture, contenant la valeur en cas de lecture.

4) Schéma d'une requête ordinaire

Une requête ordinaire peut se décomposer en 4 étapes :



1 : envoi d'une requête du processeur vers le réseau
2 : connexion et envoi d'une requête du réseau vers la mémoire.
3 : acquittement de la mémoire vers le réseau.
4 : envoi d'un acquittement du réseau vers le processeur et fin de la connexion.

Entre les étapes 1 et 2, le processeur peut éventuellement séjourner dans une file d'attente avant d'obtenir la connexion. cf gestion des conflits.

Les requêtes ordinaires n'assurent aucune cohérence dans les ordres de lecture/écriture émis par les différents processeurs.

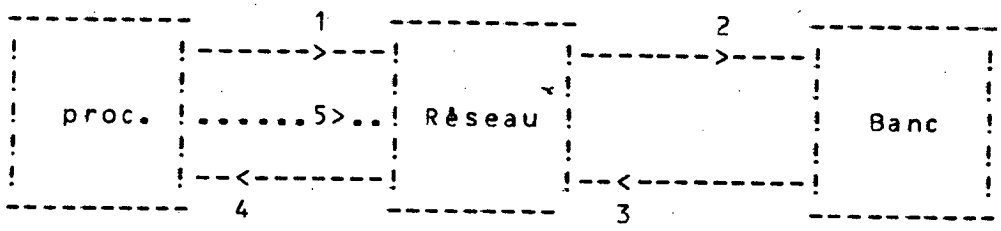
5) Schéma d'une requête privilégiée

Les requêtes privilégiées permettent de protéger l'accès aux variables partagées lors des exécutions des instructions TS, TC, WAIT.

Dès qu'un processeur obtient une connexion privilégiée pour accéder à une variable partagée, il conserve cette connexion pour modifier le contenu de la variable et renvoyer

la nouvelle valeur (éventuelle) calculée en mémoire. Aucun autre processeur n'a accès à cette variable durant tout ce temps. Seulement après avoir écrit en mémoire la valeur modifiée et après avoir fini l'exécution de l'instruction TS, TC, ou WAIT, le processeur libère la connexion.

La cohérence de ces instructions est ainsi respectée. Une requête privilégiée se décompose en étapes :



1, 2, 3, 4 : comme pour une requête ordinaire, mais la connexion est cette fois réservée à l'étape 4.
5 : envoi d'une demande de libération de connexion du processeur vers le réseau, et libération de la connexion.

CHAPITRE 4 :

ASSEMBLEUR : STRUCTURE ET SYNTAXE

I. Structure d'un programme

II. Appel d'un sous-programme

III. Retour d'un sous-programme

IV. Cas du programme principal

V. Syntaxe de l'assembleur

I. STRUCTURE D'UN PROGRAMME

1) Description

Chaque programme se décompose en 7 parties :
- déclaration du programme
- déclaration des paramètres (s'il y a lieu)
- déclaration des variables locales (s'il y a lieu)
- déclaration des constantes numériques (s'il y a lieu)
- déclaration des textes des messages (s'il y a lieu)
- corps du programme
- fin du programme.

La déclaration du programme doit être la 1ère partie.
La fin du programme doit être la dernière partie.
Les déclarations de paramètres, variables locales, constantes doivent précéder le corps du programme.

Toutes les variables utilisées doivent être déclarées. Il n'existe aucune déclaration implicite.

2) Déclaration du programme

Cette partie comprend 1 seule instruction
- de code symbolique PROG dans laquelle on spécifie
- le nom du programme,
- le mode de stockage : horizontal ou vertical ou cache.

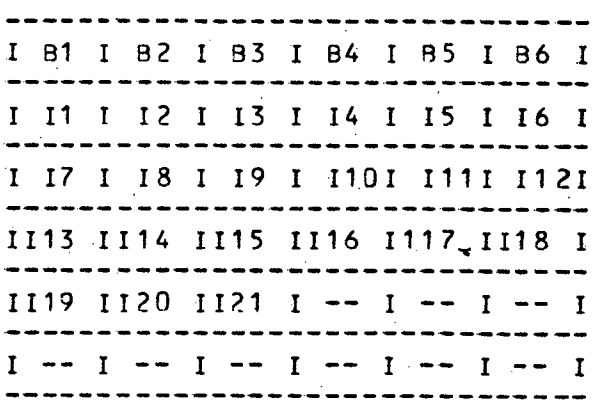
S'il est implicite, le mode est "cache". (cf. a), b), c)).

a) Stockage horizontal

Les instructions du code généré sont réparties entre les bancs-mémoire.
Elles sont numérotées à partir de 1 : I1, I2,...
Le banc m contient alors toutes les instructions de numéro i congru à m modulo M, où M est le nombre de bancs

Ii appartient à m si et seulement si, i = m modulo(M).

exemple : M = 6
pp : PROG HZ



b) Stockage vertical

Toutes les instructions du code généré sont stockées sur un seul et même banc.

Le numéro de ce banc peut être :

- un numéro absolu m
- le numéro p du processeur exécutant le code (modulo le nombre de bancs-mémoire).

Des copies locales sont alors créées dans les différents bancs concernés.

c) Stockage cache

Les instructions du code généré sont accessibles par l'un quelconque des processeurs sans pénaliser le temps d'exécution. Le temps d'accès au code est totalement ignoré, contrairement aux modes de stockage précédents, où les temps d'accès aux bancs contenant le code généré sont décomptés.

Exemples :

- test : PROG ou test : PROG CA

Le programme de nom test est accessible sans pénalité de temps.

3) Déclaration des paramètres

Cette partie comprend autant d'instructions que le programme contient de paramètres,

- de code symbolique PAR

dans lesquelles on spécifie :

- le nom du paramètre

Exemples :

- alpha : PAR

(déclaration du paramètres de nom alpha)

4) Déclaration des variables locales

Cette partie comprend autant d'instructions que le programme contient de variables locales,

- de code symbolique DEF dans lesquelles on spécifie
- le nom de la variable
- son type : entier, réel, ou sémaphore cf a)
- sa longueur : 1 pour les scalaires, > 1 pour les vecteurs (cf. b)).
- son mode de stockage : horizontal ou vertical (cf. c) et d)).
- eventuellement l'option d'initialisation : cf 6).

5) Types des données (variables ou constantes)

Les instructions manipulent trois types de données.

- type entier (IN).

Une donnée de type entier est un mot-mémoire de 36 bits.

- type réel (RE).

Une option permet de ne pas stocker dans le simulateur les données de type réel. Dans ce cas, les instructions d'arithmétique flottante sont simulées et non exécutées dans Multics. Ce choix permet un gain de place mémoire et de temps calcul dans la simulation.

- type sémaphore (SM).

Une donnée de type sémaphore est une chaine de 36 bits dont les instructions peuvent marquer chaque bit à "1"b ou "0"b. On peut utiliser un sémaphore comme un verrou constitué d'un jeton et d'une file d'attente. Le jeton est le 1er bit (à gauche), qui vaut "1"b si le jeton est pris, "0"b si le jeton est libre.

La file d'attente est représentée par les 35 autres bits. Le bit (p+1) vaut "1"b si le processeur p est dans la file d'attente, "0"b sinon. (cf. figure 19).

Les variables de type entier et réel sont lues ou écrites en mode ordinaire et ne peuvent faire l'objet de requêtes privilégiées.

Par contre, les variables de type sémaphore peuvent être utilisées en mode ordinaire et privilégié. Leur accès peut donc être protégé, ce qui assure leur cohérence lorsqu'elles sont partagées entre les processeurs. (cf. chap. 3 IV)).

b) longueur d'une variable

- variables scalaires : la longueur vaut 1.
- vecteurs : la longueur est >1

La longueur d'un vecteur est le nombre de mots alloués à ce vecteur. Un mot est repéré par son déplacement, par rapport au 1er mot du vecteur. Le déplacement est nul pour le 1er mot, il vaut 1 pour le 2e mot, etc...

Le déplacement correspond à l'indice ssi l'indexation du vecteur commence par 0.

exemple :

x DEF RE(100), Hz
x est un réel de longueur 100.
x(0) réfère au 1er élément d'indice 0.
x(99) réfère au 100e élément d'indice 99

c) Stockage horizontal

Les éléments d'un vecteur stocké horizontalement sont répartis entre tous les bancs-mémoire.

Soit x1, x2,...,xL les éléments indexés de 1 à L du vecteur x de longueur L.

Soit B1, B2,...,BM les bancs-mémoire numérotés de 1 à M.

Soit m, (0 < m < M+1) le numéro d'un banc.

Le banc m contient tous les éléments d'indice i congru à m modulo M.

x appartient a m ssi : i = m (modulo M)

Exemple : M = 5
x : DEF RE(12), HZ

B1	B2	B3	B4	B5
x1	x2	x3	x4	x5
x6	x7	x8	x9	x10
x11	x12	--	--	--

d) Stockage vertical

Tous les éléments d'un vecteur sont stockés dans le même banc-mémoire.

Le numéro de ce banc peut être :

- un numéro absolu m
- le numéro du processeur de travail p (modulo le nombre de bancs-mémoire).
- le numéro du processeur de travail p, décalé de n bancs, le tout modulo le nombre de bancs-mémoire, i.e.:

$$m = (p+n) \bmod M$$

Exemples : M = 5

- x : DEF RE(100), VE(BM1)

x est stocké sur le banc B1.

- accès : DEF TS(1), VE(PR)

accès est stocké sur le banc Bp, où p est le numéro du processeur p = 1 - B1 p = 6 - B1

- i : DEF IN(1), VE(PN2)

exécuté par	P1	->	i stocké sur		B3
"	"	P2	->	"	B4
"	"	P3	->	"	B5
"	"	P4	->	"	B1
"	"	P5	->	"	B2

6) L'option d'initialisation

Cette option permet d'initialiser la variable lors de l'activation du sous-programme, par lecture dans une table de données, préablement créée par un programme éxécuté sur l'ordinateur hôte (i.e. Multics); la gestion des tables de données s'effectue par utilisation d'un interface disponible appelable depuis FORTRAN, PL1, etc (module tabdon).

Exemple d'application : lecture d'un maillage d'éléments finis.

La syntaxe de l'option admet 3 formes :

- INIT (n)

n=index dans le catalogue

exemple :

```
tab : DEF IN(1000),HZ,INIT(5)
```

- INIT (Rr) L'index dans la table est donné par le contenu du registre r

exemple :

```
tab1:DEF IN(1000),VE(PR3),INIT(R10)
```

- INIT (Pp)

L'index dans le catalogue est donné par : numéro de processeur+p

exemple:

```
tab2:DEF IN(2000),VE(PR1),INIT(P2)
```

REMARQUES :

i) Si le calcul de l'index donne la valeur 0, les données sont initialisées par lecture au terminal (ou sur le fichier user_input).

ii) L'utilisateur gère ses tables de données par appel aux points d'entrée décrits a l'annexe B.

iii) Dans le cas du programme principal, les variables déclarées avec l'option INIT ne sont initialisées qu'une fois: elles sont en effet partagées (cf. IV).

7) Déclaration des constantes numériques

Cette partie comprend autant d'instructions que le programme contient de contraintes numériques de code symbolique CST dans lesquelles on spécifie :

- le nom symbolique de la constante
- le type (entier, réel, sémaphore) de la constante
- la valeur de la constante doit être donnée à l'appel du programme.

Exemples :

- P : CST IN,(5)

P est une constante entière P = 5

- sema : CST SM,("1"b)

sema est une constante sémaphore qui vaut "10...0"b

- N : CST IN, INIT

N est une constante entière dont la valeur est lue à l'appel du programme.

8) Déclarations des textes de messages

Cette partie comprend autant d'instructions que le programme contient de textes de code symbolique TEXTE dans lesquelles on spécifie

- le nom du texte
- sa valeur

Exemples :

d : TEXTE (début).

9) Corps du programme

Cette partie comprend toutes les instructions exécutables. Ces instructions sont groupées en cinq classes :

- arithmétique entière : opérandes entières ou sémaphores
- arithmétique flottante : opérandes réels
- logique et décalage : opérandes entiers ou sémaphores
- contrôle ordinaire : branchements et jumps
- contrôle spécial : opérandes sémaphores, traitement de sous-programmes.

La liste de ces instructions est donnée à la table.

Ces instructions peuvent être étiquetées ou non. Leur code symbolique est suivi de la zone opérande.

10) Fin du programme

Cette partie ne contient qu'une instruction
- de code symbolique END

Il n'existe qu'une instruction END par programme. C'est la dernière instruction.

11) Commentaires

Chaque partie du programme peut contenir des commentaires, qui peuvent être placés :
- au début de la partie du programme
- à la fin d'une instruction
- entre deux instructions.

12) Pseudo-instruction PAUSE

Cette disposition a pour but de faciliter à l'utilisateur la mise point de ses programmes.

Une pseudo-instruction PAUSE a pour effet de "marquer" l'instruction qui la suit; lors du traitement d'une instruction marquée, l'utilisateur a la possibilité d'interroger interactivement le simulateur sur le contenu de ses registres, la valeur des variables, l'état des processeurs, du réseau,...

Pour celà, l'utilisateur dispose de mots-clefs connus du simulateur :

Liste des mots cles : (x designe le parametre; pas de blanc entre le mot cle et le parametre)

Rx	:	imprime le contenu du registre n0. x.
markax	:	pose une marque après ("after") l'instruction de compteur ordinal x.
markbx	:	pose une marque avant ("before") l'instruction de compteur ordinal x.
markabx	:	pose une marque avant et après l'instruction de compteur ordinal x.
demarkx	:	enleve la marque posée à l'instruction de compteur ordinal x.
lm	:	liste les marques du programme.
chpx	:	change le numéro du processeur observe : permet d'obtenir l'etat du processeur n0. x.
co	:	imprime la valeur du compteur ordinal.
cc	:	imprime la valeur du code condition.
go	:	continue l'execution.

Il est possible d'interroger le simulateur soit avant, soit après une instruction.

Exemple :

```
        PAUSE AB
        ILD   R0,=0          ! interroger le simulateur avant
                             ! et après l'instruction ILD
        IST   R0,x
        PAUSE                ! "PAUSE" est équivalent à "PAUSE AB"
        IMUL  R0,R1
        PAUSE A              ! faire une "PAUSE" après
        ICP   R0,R2          ! l'éxécution de ICP
        PAUSE B              ! faire une "PAUSE" avant
        BE    suite          ! l'éxécution de BE
```

13) Option de trace des tests sur variables réelles.

Lorsque l'utilisateur demande à ne pas éxécuter les instructions flottantes, il convient dans ce cas de prévoir quel traitement effectuer quand une instruction FCP (comparaison entre flottants) est rencontrée. En effet cette instruction modifie normalement le Code-Condition et donc éventuellement le déroulement du programme; enoutre il est souhaitable que l'utilisateur puisse se servir du même programme avec ou sans l'option d'éxécution des instructions flottantes.
La valeur que le Code-Condition (-1,0,+1) doit prendre dans ce cas est définie par l'utilisateur: celui-ci donne pour chaque processeur une trace d'éxécution donnant les valeurs successives de (CC+1). Les traces de chaque processeur sont des suites de caractères 0, 1, 2, regroupées dans un même fichier; les différentes traces sont séparées par le caractère 3. Les traces peuvent être générées par l'utilisateur, soit au moyen d'un programme pl1, fortran,... soit au moyen d'un editeur de texte (Les caractères blancs, saut de page, tabulation dans la trace sont ignorés)

II. APPEL D'UN SOUS-PROGRAMME

1) Instructions CALL et ARG

Pour appeler un sous-programme, on utilise l'instruction:
- de code symbolique CALL dans laquelle on spécifie
- le nom du sous-programme appelé et que l'on fait suivre des instructions
- de code symbolique ARG dans lesquelles on spécifie
- le nom des arguments
- éventuellement, le déplacement si l'argument est un élément de vecteur.

Ce groupe d'instructions a pour effet de :
- transmettre les arguments
- allouer les variables locales du sous-programme
- sauve
- créer le nouvel environnement relatif au sous-programme.

2) Transmission des arguments

La liste des arguments transmis correspond, dans le même ordre, à la liste des paramètres du sous-programme.
Tous les types sont autorisés.
L'argument peut être un scalaire, un vecteur complet, ou un élément de ce vecteur (on indique alors le déplacement).
Les arguments sont transmis par adresse. (non par valeur). Les adresses sont rangées dans la table des symboles du sous-programme.
Les paramètres héritent des caractéristiques des arguments associés :
- le type reste le même;
- la longueur d'un vecteur aussi (si un seul élément est transmis, le paramètre est un scalaire);
- le mode de stockage est, en général, le même. Cependant, l'élément d'un vecteur stocké horizontalement acquiert un mode de stockage vertical, dont le numéro de banc m dépend du déplacement d et du nombre de bancs M :

$$m = d+1 \text{ (modulo } M)$$

Les paramètres d'un sous-programme peuvent être les arguments d'un second sous-programme.

3) Allocation des variables locales

Les variables locales déclarées dans le sous-programme sont allouées en mémoire. A chaque variable est associée son adresse en mémoire (cf. chap 1 III).
Les adresses des variables locales sont rangées dans la table des symboles du sous-programme.

4) Sauvegarde du programme appelant

Le programme appelant est sauvegardé dans la pile du sous-programme. Il est empilé sur la dernière sauvegarde. L'élément de pile associé contient :

- l'adresse de début du programme appelant;
- sa table des symboles;
- le compteur ordinal de retour;
- les contenus des 16 registres de calcul.

Par contre, le code condition n'est pas sauvegardé.

5) Environnement du programme appelé

La référence au programme appelé est rangée dans le haut de pile du processeur, qui contient :

- l'adresse de début du sous-programme;
- sa table des symboles;

L'état du processeur est modifié :

- le compteur ordinal est positionné à 1;
- par contre, le contenu des registres et le code condition sont, au lancement du sous-programme, ceux du programme appelant.

III. RETOUR D'UN SOUS-PROGRAMME

1) Instruction RETURN

La fin d'un sous-programme est exécutée par l'instruction
- de code symbolique RETURN.

Il peut exister plusieurs instructions RETURN par sous-programmes, mais il en existe au moins une.

La 1ere instruction RETURN exécutée a pour effet de :
- libérer la place mémoire des variables locales
- restituer l'environnement du programme appelant.

2) Libération des variables locales

La place mémoire de toutes les variables locales est récupérée. Leurs adresses sont disponibles pour de nouvelles allocations.

3) Retour au programme appelant

La sauvegarde du programme appelant est retirée de la pile, elle est rangée dans le haut de pile, qui contient alors :
- l'adresse de début du programme appelant;
- sa table des symboles;

L'état du processeur est restauré :
- le compteur ordinal est positionné à la valeur de retour sauvegardée.
- les contenus des registres sauvegardés sont restitués.
- par contre, le code condition garde la valeur qu'il avait à la fin du sous-programme.

IV. CAS DU PROGRAMME PRINCIPAL

1) Environnement du programme

La référence au programme principal est rangée dans tous les processeurs, les hauts de pile des processeurs contiennent donc :

- l'adresse de début du programme principal
- sa table des symboles.

Les variables locales du sous-programme ne sont allouées qu'une fois. Elles deviennent donc partagées entre les processeurs qui connaissent tous les adresses de ces variables (cf. II 3).

2) Retour du programme

La fin du programme principal s'exécute par l'instruction de code symbolique RETURN.

Il peut exister plusieurs RETURN (au moins 1).

Les autres éléments de la pile (autres que le haut de pile contenant le programme principal) sont vides.

L'exécution s'arrête donc. Le processeur a terminé. Les variables locales ne sont pas libérées. En effet, tous les processeurs ne terminent pas en même temps ; les variables doivent rester allouées jusqu'à la fin d'exécution de tous les processeurs.

V. SYNTAXE DE L'ASSEMBLEUR

1) Règles générales

<nom> : <CODE SYMBOLIQUE> <zone opérande>.

On y distingue 3 zones :

- au début, un nom ou une étiquette peut (ou non) figurer
- Il est suivi du code symbolique.
- La 3e zone est la zone opérande, qui peut être vide.

exemple :

```
pp : PROG
     début : IADD R0, R1
```

a) Zone nom

Le nom peut être :

- un nom de programme
- un nom de variable (paramètre, variable locale, constante)
- une étiquette.

Un nom est un mot de 6 caractères alphanumériques au plus.
Le 1er caractère ne peut être R.
Il ne peut contenir les caractères ! =

b) zone opérande

Les diverses zones opérandes sont décrites ci-après. Elles peuvent contenir 0, 1 ou 2 opérandes.
Le séparateur des opérandes est sur le caractère ; les blancs sont ignorés.

c) séparateur des zones

- Le séparateur des zones <nom> et <code symbolique> est le caractère : . Les blancs sont ignorés.
- Le séparateur des zones <code symbolique> et <opérandes> est le caractère blanc. Il peut exister plusieurs blancs.

2) Syntaxe des déclarations de programme

```
<pgm> : PROG   <stockage>
```

où :

- <pgm> est le nom du programme
- <stockage> est le mode de stockage :

```
HZ  -  horizontal
VE  -  vertical VE(BMm) - sur le banc m
VE(PR)  - sur le banc p
VE(PNn) - sur le banc (p+n)
(p désigne le numéro du processeur de travail)
CA      -     cache
rien
```

exemples :

```
pp : PROG CA ou pp : PROG
sp : PROG VE(PR)
```

3) Syntaxe des déclarations de paramètres

<par> : PAR

où :

- par est le nom du paramètre
  exemple : alpha : PAR

4) Syntaxe des déclarations de variables locales

<var> : DEF <type> (<lg>), <stk>,<optINIT>

où

- <var> est le nom de la variable
- <type> est le type de la variable :
  IN type entier
  RE type réel (flottant)
  SM type sémaphore.

- <lg> est le nombre de mots alloués à la variable (entre parenthèses).

Aucun blanc ne sépare le type et la longueur. Longueur vaut 1 pour les scalaires et les sémaphores.

- <stk> est le mode de stockage de la variable :
  HZ Horizontal
  VE Vertical
  VE(BMm) sur le banc m
  VE(PR) sur le banc p
  VE(PNn) sur le banc (p+n) modulo M

où p est le numéro du processeur de travail, M est le nombre de bancs-mémoire.

- <optINIT> : option d'initialisation de la variable, éventuellement absente (cf ch 4,I.6, et annexe B):

<optINIT> ::= INIT (<I>)

où <I> permet de retrouver le tableau dans la table :
  <I> = n = entier de 1 a 100 = index du tableau dans le catalogue;
  <I> = Rr : l'index du tableau est donné par le contenu du registre r lors de l'activation du sous-programme;
  <I> = Pn : l'index du tableau est donné par : numéro du processeur + n

Si l'on obtient pour l'index la valeur zero, les valeurs sont initialisées par lecture au terminal.

exemples :

```
x : DEF RE(100), HZ
prot : DEF SM(1), VE(BM1)
```

5) Syntaxe des déclarations de constantes numériques

<cst> : CST <type>, (<valeur>)

ou:

<cst> : CST <type>, INIT

où :

- cst est le nom de la constante

- type est le type de la constante
  IN type entier
  RE type réel
  SM type sémaphore (chaine de 36 bits).
- valeur est la valeur de la constante
- INIT signifie que la valeur sera donnée à l'appel du programme.

Exemples :

```
P : CST  IN, (5)
sema : CST   SM, ("101"b)
x4 : CST   RE, (4.)
N : CST   IN, INIT
```

6) Syntaxe des déclarations de messages

```
<t> : TEXTE  (<valeur>)
```

où :

- t est le nom du message
- valeur est son contenu, indiqué entre parenthèses, séparé par au moins un blanc du code symbolique TEXTE, et contenant au plu 8 caractères autres que !

Exemples :

```
t1 : TEXTE (DEBUT)
t2 : TEXTE (FIN).
```

7) Syntaxe des instructions exécutables

```
<étiq> : <CODE>  op1, op2
```

où :

- étiq est le nom d'une étiquette. Elle peut ne pas figurer.
- CODE est le code symbolique de l'instruction
- op1 et op2 sont, respectivement, les 1er et 2e opérandes.

Ils peuvent ne pas figurer.

Exemple :

```
début : FMUL R1,X(R2)
FADD R1,R3
```

Les syntaxes diffèrent suivant la classe des instructions.

a) syntaxe des instructions d'arithmétique entière

```
<étiq> : I ... Rr, op2
```

ou bien :

```
I ... Rr, op2
```

où :

- étiq est le nom d'une étiquette
- I... est le code symbolique :
  IAAD, ISUB, IDIV, IMUL, ILD, IST, ICP
- Rr désigne le 1er opérande : registre de numéro r

- op2 désigne le 2e opérande cf. c).

b) syntaxe des instructions d'arithmétique flottante

<étiq> : F... Rr, op2

ou bien :

F... Rr, op2

cf. a) avec, pour code :

FADD, FSUB, FDIV, FMUL, FLD, FSTQ, FCP.

c) syntaxe de op2

SYNTAXE	OPERANDE
Rs	Registre de numéro s
=p	Opérande immédiat de valeur p
c	Constante de nom c
x	Scalaire de nom x
x(d)	Elément du vecteur de nom x avec déplacement de valeur d
x(Rd)	Elément du vecteur de nom x avec déplacement dans le registre de numéro d

d) syntaxe des instructions de logique

- instructions LAND, LOR, LOX (2 opérandes), LCP, LMCP

<étiq> : L ... Ø Rr, op2

ou bien :

L ... Ø Rr, op2

cf. a) et c)

- instruction LNO (1 opérande)

<étiq> : LNO Rr

ou

LNO Rr

où Rr est le registre de numéro r.

- instructions LDD, LDG (1 opérande + 1 décalage).

<étiq> : LD. Rr, = p

ou

LD. Rr, = p

où :

- Rr est le registre de numéro r (opérande)
- p est la valeur du décalage.

e) Syntaxe des instructions de contrôle ordinaire

- branchements : B, BL, BE, BG, BLE, BNE, BGE.

<étiq> : B

B

<étiq> : B .. <label>

B .. <label>

où:

<label> désigne le nom d'une étiquette associée à une instruction du programme.

- jumps : J, JL, JE, JG, JLE, JNE, JGE

<étiq> : J ...

J ...

f) Syntaxe des instructions de contrôle spécial

- instructions TS, TC (T.)

<étiq> : T. op

ou

T. op

où op désigne le nom d'une variable de type sémaphore

- instruction WAIT

<étiq> : WAIT Rr, op

ou

WAIT Rr, op

où Rr désigne le registre de numéro r et op désigne le nom d'une variable de type sémaphore.

- Instruction NB

<étiq> : NB, P

<étiq> : NB, M

NB, P

NB, M

où P signifie nombre de processeurs, M signifie nombre de bancs-mémoire.

- instruction PRN

<étiq> : PRN, Rr

ou bien :

<étiq> : PRN, Rv, V

ou bien :

PRN, Rr

ou bien :

PRN, Rr, V

où :

Rr est le registre de numéro r.

V signifie valeur (cf. jeu d'instructions).

- instruction CALL

<étiq> : CALL sp

ou bien :

CALL sp

où sp désigne le nom du sous-programme.

- instruction ARG

<arg> : ARG

ou bien :

<arg> : ARG (d)

ou bien :

<arg> : ARG (Rd)

où :

<arg> est le nom de l'argument.

d est le déplacement éventuel (valeur)

Rd est le registre (numéro d) contenant le déplacement éventuel.

- instruction RETURN

<étiq> : RETURN

ou bien :

RETURN

8) syntaxe des instructions de mesure

- instruction STAT
  STAT D
  STAT F

D : Début des mesures statistiques
F : Fin des mesures statistiques

- instruction MESS
  MESS t, <op>

t : nom du texte du message
<op> : opérande dont le message imprime la valeur
  * <op> = Rr registre r
  * <op> = x variable scalaire de nom x card
  * <op> = x(d) élément du vecteur x avec déplacement d
  * <op> = x(Rd) élément du vecteur x avec déplacement dans le registre d
  * <op> = c constante de nom c

- instruction ETAT
  ETAT <niveau>
  ETAT <niveau>, DT = <dt>
  ETAT <niveau>, DK = <dk>

<niveau> : niveau d'impression (7, 8, 9, ou 10)
<dt> : intervalle de temps entre 2 impressions
<dk> : intervalle d'événements entre 2 impressions

CHAPITRE 5

OUTILS DE MESURE DANS MUPI

I. Statistiques

II. Messages

III. Etat de la machine

I. STATISTIQUES

Pour mesurer les performances de la machine et d'un algorithme, on peut faire imprimer des statistiques concernant d'une part les processeurs, d'autre part les bancs-mémoire.

Les statistiques relatives aux processeurs indiquent la vitesse d'exécution, le temps passé en attentes sur le réseau (requêtes mémoire) et en arrêt sur le BIP (attente d'une relance par un autre processeur).

Les statistiques relatives aux bancs indiquent le taux d'occupation, les conflits apparus.

1) Instruction_STAT

Elle permet de déterminer le début et la fin des mesures :
- -STAT D : Début de mesures statistiques
- -STAT F : Fin de mesures statistiques.

Si aucune instruction STAT F n'est exécutée après une instruction STAT D, les mesures durent jusqu'à l'arrêt complet de la machine, ce qui permet d'arrêter les mesures à un instant bien défini.

- Impressions :

STAT : nbp = ... nbm = ... de t1 = ... a t2 = ... durée = ...

nbp : nombre de processeurs
nbm : nombre de bancs-mémoire
t1 : heure du début des mesures (instruction STAT D)
t2 : heure de fin des mesures (instruction STAT F)
durée : durée des mesures (t2-t1)

2) Statistiques_relatives_aux_processeurs

Pour chaque processeur, la première ligne indique la répartition du temps de travail dans le processeur, en trois catégories :
- temps passé en requêtes mémoire sur le réseau
- temps passé en attente de relance ou blocage sur le BIP
- temps d'exécution proprement dit.

Les lignes suivantes indiquent le temps passé dans chaque file d'attente des bancs-mémoire (par suite de conflit d'accès à ce banc).

1ère ligne :
SPp :
RQT:t=...%=...n=...BLO:t=...%=...n=...PRO:t=...%...n=...
SPp : statistiques relatives au processeur p.
RQT : statistiques relatives aux requêtes mémoire
t=... temps total passé en attente d'acquittement

%=... pourcentage par rapport à la durée totale des mesures
n=... nombre de requêtes émises.
BLO : statistiques relatives au BLOcage sur le BIP
t=... temps total passé en attente de relance
%=... pourcentage
n=... nombre de blocages en attente de relance sur le BIP
PRO : statistiques relatives à l'exécution du PROcesseur
t=... temps total d'exécution proprement dite
(requêtes-mémoire et blocages sur BIP décomptés).
%=... pourcentage
n=... nombre d'instructions d'arithmétique flottante
exécutée.
lignes suivantes :
SP p : A m : t = ... % = ... n = ...
SP p : statistiques relatives au processeur p.
A m : statistiques relatives à la file d'attente du banc m.
t=... temps total passé par le processeur p dans la file
d'attente du banc m
%=... pourcentage/durée totale des mesures
n=... nombre de séjours dans la file d'attente du banc m.
Sur chaque ligne, on indique les résulats relatifs à 2
bancs.

3) Statistiques relatives aux bancs-mémoire

Pour chaque banc-mémoire, la première ligne indique le
taux d'occupation du banc, en distinguant :
- le temps où le banc est libre (aucune connexion)
- le temps où le banc est connecté sans conflit (file
d'attente associée vide).
- le temps où le banc est connecté avec conflit (file
d'attente associée occupée).

Les lignes suivantes indiquent la durée des conflits entre
2, 3,...P processeurs. (P est le nombre de processeurs
configurés).

1ère ligne :
SBm : LIB :
t=...%=...n=...OSC:t=...%=...n=...OAC:t=...%=...n=...

LIB : temps où le banc est LIBre (pas de connexion).
OSC : temps où le banc est Occupé Sans Conflit (file
d'attente vide).
OAC : temps où le banc est Occupé Avec Conflit (file
d'attente occupée).
t=... t : temps total
%=... : pourcentage/durée des mesures
n=... : nombre de fois où le banc se trouve dans cet état.

Lignes suivantes

SB m:C p:t =... % =... n =...

SB m : statistiques relatives au Banc m
C p : statistiques relatives au conflit avec p processeurs (p>1)
t = ... temps total des conflits à p processeur
% = ... pourcentage/durée des mesures
n = ... nombre de conflits à p processeurs.

Sur chaque ligne, on indique les résultats relatifs à 2 conflits.

II. MESSAGES

A tout instant, un processeur peut faire imprimer un message qui permet de localiser dans le temps les séquences d'instruction, et de vérifier les valeurs de certains registres ou certaines variables.

Par exemple : un message en fin de programme détermine la durée d'exécution de ce programme. Des messages avant un point de synchronisation permettent d'en vérifier l'efficacité.

Instruction MESS

L'instruction MESS réalise ces impressions de messages.

- 1er opérande : nom du message (cf. chapitre 4)
- 2e opérande : il peut être :
  - * un registre Rr
  - * une variable scalaire
  - * un élément de vecteur
  - * une constante symbolique
- ne modifie pas l'état du processeur.
- durée d'exécution nulle. N'envoie pas de requête au réseau.
- impression :

```
MESS:H=<heure> Pp PROG:<nom>TEXTE:<texte><var>=<valeur>
```

heure : heure d'émission du message
Pp : numéro p du Processeur émetteur
PROG:<nom> : nom du PROGramme exécuté
TEXTE:<texte>: texte du message (1er opérande).
<var> = <valeurs> : nom et valeur du 2e opérande.

Si l'opérande est une chaine de 36 bits, on imprime les 36 bits.

Exemples :

```
   pp : PROG
        ...
        t1 : TEXTE (début)
        ...
        ILD R0, = 1
        MESS t1, R0

MESS : H = 20 P1 PROG : pp TEXTE : début R0 = 1

  sp : PROG
        ...
        t : TEXTE (solution)
        ...
        ILD R1, = 1
        ...
        IST R1, x(R1)
        MESS t, x(R1)
MESS :H = 100 P2 PROG : sp TEXTE : solution x(1)= 1.
```

III. ETAT DE LA MACHINE - LISTE D'EVENEMENTS

Le simulateur contrôle le fonctionnement de la machine grâce à une liste d'évévements.

Pour envoyer une requête par exemple, un processeur insère un événement dans la liste.

Les événements sont datés et rangés dans la liste par ordre de date croissante, et à dates égales par ordre d'arrivée.

Le simulateur les réalise dans cet ordre.

Avant la réalisation d'un événement, on peut, grâce à l'instruction ETAT (cf. III.1) faire imprimer, l'état actuel de la machine, qui contient :

- l'événement lui-même
- l'état du réseau
- l'état de chaque processeur

Ces impressions permettent de contrôler le déroulement des programmes sur chaque processeur.

1) Instruction ETAT

L'instruction ETAT permet de faire imprimer l'état de la machine, avec les options suivantes :

-DT = dt : impressions toutes les dt unités de temps
-DK = dk : impressions tous les dk événements
-implicite : impressions tous les événements.

Les impressions sont plus ou moins détaillées suivant le niveau choisi :

- niveau = 10 : impressions de tous les événements, et à chaque intervalle de mesure, impressions du réseau et de tous les processeurs.
- niveau = 9 : à chaque intervalle de mesure, impression de l'événement qui va se réaliser, du réseau et de tous les processeurs juste avant cet événement.
- niveau = 8 : à chaque intervalle de mesure, impression de l'événement qui va se réaliser, du réseau et du processeur concerné (état précédent l'événement).
- niveau = 7 : à chaque intervalle de mesure, impression de l'événement qui va se réaliser.

syntaxe : ETAT <niveau>
ETAT <niveau>,DT = <dt>
ETAT <niveau>,DK = <dk>

Ex : ETAT 10
ETAT 8,DT=80

2) Impression d'un événement

EVT:H = <heure> <type> Pp Bm <rqt> S = <ad.S>

heure : date de réalisation de l'événement
type : type de l'événement : cf liste ci-dessous
Pp : p = numéro du processeur concerné
Bm : m = numéro du banc-mémoire concerné
S : adresse dans Multics (Buddy-System)
(S=0 pour les variables flottantes)
rqt : type de la requête :
E écriture
L lecture

Types d'événements :

PL : processeur libéré (par un TC)
PP : processeur forêt (fin de l'instruction)
PAO : le processeur recoit un acquittement de requête ordinaire
PAP : le processeur recoit un acquittement de requête privilégiée.
BRO : le banc mémoire recoit une requête ordinaire
BRP : le banc mémoire recoit une requête privilégiée
RRO : le réseau recoit une requête ordinaire
RRP : le réseau recoit une requête privilégiée
RAO : le réseau recoit un acquittement de requête ordinaire
RAP : le réseau recoit un acquittement de requête privilégiée
RRL : le réseau recoit une requête de libération du mode privilégié.

ex : EVT : H = 500 BRO P1 B1 L S = 0

3) Impression de l'état du réseau

RES : H = <heure> P : ..... B : .....

heure : date de réalisation de l'événement
P : connexions des processeurs, dans l'ordre : P1, P2,... (cf. ci-dessous).
B : connexions des bancs-mémoire, dans l'ordre : B1, B2,... (cf. ci-dessous).

Connexions des processeurs

L : libre (le processeur n'est connecté à aucun banc)
O m : le processeur est connecté en mode ordinaire au banc m
A m : le processeur attend une connexion avec le banc m
P m : le processeur est connecté en mode privilégié avec le banc m
R m : le processeur a réservé l'accès au banc m.

Connexions des bancs

L : le banc est libre
O : le banc est occupé.

Exemple : RES : H = 500 P : 01 A1 L B : 0 L L

4) Impression de l'état d'un processeur

Pp : H = <h><sp><m> CC=<cc> CO=<nom>!<CO> <CS><sc>!!Rr Nn Dd Tt

<h> : heure interne au processeur
<sp> : situation-présente :
L : Libre (fin d'instruction)
A : Attend (requête mémoire en cours)
B : Bloqué (attend une relance sur BIP)
T : Travaille (instruction en cours)
m : mode de travail :
0 : Ordinaire
P : Privilégié
CC = <cc> : Code Condition : -1,0,+1
CO = : Compteur Ordinal
nom : nom du programme éxécuté
co : valeur du compteur ordinal
CS : Code Symbolique de l'instruction
sc : numéro du sous-cycle en cours
Rr : numéro de registre
Nn : numéro dans la table des symboles
Dd : déplacement
Tt : Type d'adressage

exemple :
P1 : H = 300 A0 CC = 0 CO = pp!3!!FLD 6!R1 N1 D0 T0

5) Code opération

L'assembleur de Muppy traduit les instructions du fichier source en chains de 36 bits qu'il stocke dans le fichier objet.
Chaque chaine de 36 bits contient 5 zones :

code op	n-reg	n-symbole	déplacement	type-ad
7bits	4bits	7bits	16bits	2bits

a) code-op

* Les 3 premiers bits donnent la chaine d'instructions.
* Les 4 bits suivants donnent le numéro interne à la classe (cf. glossaire).

b) n-reg

* Pour les instructions avec un 1er opérande, il contient le numéro de registre de cet opérande (qui sera aussi celui du résultat éventuel).
* Pour l'instruction MESS, il contient le numéro dans la table des textes du message à imprimer.
* Pour l'instruction ETAT, il convient le niveau.
* Pour toute autre instruction, n-reg est nul.

c) n-symbole

* Pour les instructions avec un 2e opérande symbolique ou un sémaphore, n-symbole est strictement positif et contient le numéro dans la table des symbole de l'opérande ou sémaphore.
* Pour les instructions de branchement, n-symbole est strictement positif et contient le numéro dans la table des étiquettes du label.
* Pour l'instruction CALL, n-symbole est >0 et contient le numéro dans la table des sous-programmes appelés.
* Pour les instructions avec un 2e opérande constant, n-symbole est < 0 et contient le numéro dans la table des symboles.

d) Déplacement

* Pour les instructions avec un 2e opérande dans un registre, déplacement contient le numéro de ce registre.
* Pour les instructions avec un 2e opérande immédiat, déplacement contient la valeur de cet opérande.
* Pour les instructions avec un 2e opérande symbolique de longueur > 1 avec déplacement immédiat, déplacement contient la valeur de ce déplacement.
* Pour les instructions avec un 2e opérande symbolique de longueur > 1 avec déplacement dans un registre, déplacement contient le numéro de ce registre.
* Pour l'instruction CALL, déplacement contient le nombre d'instructions ARG qui suivent le CALL (remarque : ce nombre doit être égal au nombre d'instructions PAR du sous-programme appelé.
* Sinon déplacement est nul.

e) type d'adressage

* Pour les instructions avec un 2e opérande symbolique de longueur > 7 avec déplacement immédiat, type-ad vaut "00"
* Pour les instructions avec un 2e opérande symbolique de longueur > 7 avec déplacement dans un registre type-ad vaut "01"
* Pour les instructions avec un 2e opérande dans un registre et pour l'instruction PRN (dans l'option, V), type-ad vaut "10"
* Sinon type-ad vaut "11".

6) Problèmes de cohérence

A un instant donné, les informations imprimées sur l'état d'un processeur ne sont pas forcément cohérentes.

En effet, l'état interne d'un processeur varie au cours d'une instruction et n'est déterminé qu'en fin de chaque instruction, par l'architecture et le jeu d'instructions. Par contre, au niveau d'un sous-cycle, des changements d'état provisoires peuvent aboutir à une structure incohérente en apparence (dépendant de la division des instructions en sous-cycles).

Exemples d'états incohérents :

- L'horloge interne du processeur n'est pas réglée sur l'horloge de la machine : le processeur attend un acquittement de requête ou un signal de relance ; il recalera alors son horloge.

- Le compteur ordinal et le code instruction ne coincident pas : le processeur a incrémenté son compteur ordinal mais n'a pas encore décodé l'instruction correspondante.

- Le code condition et le code instruction ne coincident pas : le processeur n'a pas encore exécuté la modification du code condition exigée par l'instruction.

Le numéro du sous-cycle imprimé (sous-cycle qui va être exécuté) permet de lever les causes d'incohérence apparente (cf. liste des sous-cycles).

ANNEXE A : RECAPITULATION DU JEU D'INSTRUCTIONS

INSTRUCTIONS DE DECLARATIONS

ZONE ETIQUETTE	CODE SYMBOLIQUE	ZONE OPERANDE	SIGNIFICATION p: NUMERO DU PROCESSEUR M: NOMBRE DE BANCS-MEMOIRE
<nom>:	PROG	-	DECLARATION D'UN PROGRAMME
<nom>:	DEF	<type>(<lg>),<sto> <type>(<lg>),<sto>,INIT(<init_opt>) DECLARATION D'UNE VARIABLE LOCALE <type>: TYPE DE LA VARIABLE : IN : ENTIER RE : REEL SM : SEMAPHORE <lg> : LONGUEUR : 1 : SCALAIRE >1 : VECTEUR DE DIMENSION lg <sto> : TYPE DE STOCKAGE HZ : HORIZONTAL VE(BMm) : VERTICAL SUR LE BANC m VE(PR) : VERTICAL SUR LE BANC p MODULO M VE(PNn) : VERTICAL SUR LE BANC : (p+n) MODULO M <init_opt>:OPTION D'INITIALISATION INIT(n) : TABLEAU DE NUMERO n DANS LE CATALOGUE (n=0: SUR user_input) INIT(Rr) : INDEX = [Rr] INIT(Pn) : INDEX = (p+n)	

ZONE ETIQUETTE	CODE SYMBOLIQUE	ZONE OPERANDE	SIGNIFICATION p: NUMERO DU PROCESSEUR M: NOMBRE DE BANCS-MEMOIRE
<nom> :	PAR	-	DECLARATION D'UN PARAMETRE
<nom> :	CST	<type>,(<valeur>) ou <type>, INIT	DECLARATION D'UNE CONSTANTE type : IN, RE, SM valeur : VALEUR DE LA CONSTANTE INIT : VALEUR INITIALISEE A L'APPEL DU SOUS-PROGRAMME
<nom> :	TEXTE	(<texte>)	DECLARATION D'UN TEXTE DE MESSAGE <texte> : TEXTE DU MESSAGE

INSTRUCTIONS D'ARITHMETIQUE ENTIERE OU FLOTTANTE

CODE SYMBOLIQUE	ZONE OPERANDE	OPERATION I...SUR DES ENTIERS OU SM F...SUR DES REELS	CO	CC MODIFIE
IADD FADD	Rr, op2	ADDITION Rr <- Rr + op2	+1	NON
ISUB FSUB	Rr, op2	SOUSTRACTION Rr <- Rr - op2	+1	NON
IMUL FMUL	Rr, op2	MULTIPLICATION Rr <- Rr * op2	+1	NON
IDIV FDIV	Rr, op2	DIVISION Rr <- Rr / op2	+1	NON
ILD FLD	Rr, op2	CHARGEMENT DE REGISTRE Rr <- op2	+1	NON
IST FST	Rr, var	RANGEMENT EN MEMOIRE var <- Rr	+1	NON
ICP FCP	Rr, op2	COMPARAISON SI Rr<op2 ALORS cc<0 SI Rr=op2 ALORS cc=0 SI Rr>op2 ALORS cc>0	+1	OUI

INSTRUCTIONS DE LOGIQUE & DECALAGES

CODE SYMBOLIQUE	ZONE OPERANDE	OPERATIONS SUR DES ENTIERS OU SM	CO	CC MODIFIE
LOR	Rr, op2	OU LOGIQUE Rr <- Rr OU op2	+1	NON
LAND	Rr, op2	ET LOGIQUE Rr <- Rr ET op2	+1	NON
LOX	Rr, op2	OU EXCLUSIF LOGIQUE Rr <- Rr OU EX op2	+1	NON
LNO	Rr	NOT LOGIQUE Rr <- NOT Rr	+1	NON
LCP	Rr, op2	COMPARAISON SI Rr=op2 ALORS CC=0 SINON ALORS CC>0	+1	OUI
LMCP	Rr, op2	COMPARAISON AVEC MASQUE SI (RrETop2)=Rr ALORS CC=0 SINON ALORS CC > 0	+1	OUI
LDD	Rr, =p	DECALAGE A DROITE DE p BITS DANS Rr	+1	NON
LDG	Rr, =p	DECALAGE A GAUCHE DE p BITS DANS Rr	+1	NON

INSTRUCTIONS DE BRANCHEMENTS & JUMPS

CODE SYMBOLIQUE	ZONE OPERANDE	MODIFIE LE CO SELON LE CC			CC MODIFIE
		CC < 0	CC = 0	CC > 0	
B	label	LABEL	LABEL	LABEL	NON
BL	label	LABEL	+1	+1	NON
BE	label	+1	LABEL	+1	NON
BG	label	+1	+1	LABEL	NON
BLE	label	LABEL	LABEL	+1	NON
BNE	label	LABEL	+1	LABEL	NON
BGE	label	+1	LABEL	LABEL	NON
J	-	+2	+2	+2	NON
JL	-	+2	+1	+1	NON
JE	-	+1	+2	+1	NON
JG	-	+1	+1	+2	NON
JLE	-	+2	+2	+1	NON
JNE	-	+2	+1	+2	NON
JGE	-	+1	+2	+2	NON

SYNTAXE	SOUS-PROGRAMMES	CO	CC MODIFIE
CALL spgm	APPEL DU SOUS-PROGRAMME spgm . TRANSMISSION DES ARGUMENTS . ALLOCATION DES VARIABLES LOCALES . SAUVEGARDE DE L'APPELANT DANS LA PILE . CHARGEMENT DE L'APPELE DANS LE HAUT DE PILE	1: DEBUT DE spgm	NON
X : ARG	. VARIABLE X (TOUT LE VECTEUR ARGUMENT DE spgm	-	-
X : ARG(d)	. VARIABLE X (ELEMENT DONT LE DEPLACEMENT VAUT d) ARGUMENT DE spgm		
X :ARG(Rd)	. VARIABLE X (ELEMENT DONT LE DEPLACEMENT EST DANS LE REGISTRE d) ARGUMENT DE spgm		
RETURN	RETOUR D'UN SOUS PROGRAMME . LIBERATION DES VARIABLES LOCALES . DERNIERE SAUVEGARDE RETIREE DE LA PILE ET CHARGEE DANS LE HAUT DE PILE (REGISTRES RESTITUES)	RET. INST. APRES CALL ARG..	NON

CODE SYMBOLIQUE	ZONE OPERANDE	SYNCHRONISATIONS p:NUMERO DU PROCESSEUR	CO	CC MODIFIE
TS (Test and Set)	sema	SI 1eBIT="0"b (JETON LIBRE) ALORS .1eBIT<-"1"b (PRENDRE LE JETON) .CC = +1 .CONTINUER	+1	OUI
		SI 1eBIT="1"b (JETON PRIS) ALORS : .BIT(p+1)<-"1"b (S'INSCRIRE EN FILE D'ATTENTE) .CC = 0 .ATTENDRE UN SIGNAL DE RELANCE SUR BIP.	+0	OUI
TC (Test and Clear)	sema	SI 1eBIT="0"b ALORS ERREUR SI 1eBIT="1"b (JETON PRIS) ALORS : . SI BIT(p+1)="1"b ENVOYER UN SIGNAL DE RELANCE SUR BIP AU PROCESSEUR p .METTRE sema à"0..0"b (RENDRE LE JETON ET VIDER LA FILE D'ATTENTE)	+1	NON
WAIT	Rr,sema	SI 1eBIT="0"b (JETON LIBRE) OU SI MASQUE NON ADAPTE ALORS . CC = +1 . CONTINUER		
		SI 1eBIT="1"b (JETON PRIS) ET SI MASQUE ADAPTE ALORS . CC = 0 . ATTENDRE UN SIGNAL DE RELANCE SUR BIP		

CODE SYMBOLIQUE	ZONE OPERANDE	IMPRESSIONS
STAT	D ou F	D : Début F : Fin
MESS	t, op	heure, texte t, valeur de l'opérande op.
ETAT	niveau, DT=<dt>, DK=<dk>	état de la machine, suivant le niveau et l'intervalle d'impression (dt ou dk)

CODE SYMBOLIQUE	ZONE OPERANDE	SIGNIFICATION	CO	CC MODIFIE
NB	Rr,P	nbp : nb de processeurs Rr <- nbp	+1	NON
NB	Rr,M	nbm : nb de bancs mémoire Rr <- nbm	+1	NON
PRN	Rr,V	p : no du processeur (0 < p < ou=nbp) Rr <- p	+1	NON
PRN	Rr	p : no du processeur (0 < p < 36) bit(p+1)de Rr <-"1"b autres bits de Rr <-"0"b	+1	NON

ANNEXE 3 : GESTIONNAIRE DE TABLES DE DONNEES

But :

Enregistrer plusieurs tables de données dans un seul segment, utilisables par l'option INIT de l'instruction DEF (cf. chapître 4, par. I.6 et V.4) Ceci permet de definir des tableaux (par ex. des maillages) en FORTRAN ou PL1, et les utiliser ensuite dans MUPI.

Dans le catalogue d'une table, on note pour chaque tableau:

* le nombre de mots;
* le format (0:entiers; 1: flottants; 2 : chaînes de 36 bits);
* un commentaire choisi par l'utilisateur.

L'utilisateur peut retrouver ces informations par les commandes lister_catalogue et lister; le catalogue d'une table ne peut contenir plus de 100 entrées.

L'appel normal de la procédure se fait par l'un des points d'entrée suivants: ajouter, effacer, lister, lister_catalogue, lire_tab, comprimer.

Toutes les entrées SAUF ajouter et lire_tab peuvent etre appelées au choix comme des commandes (en interactif) ou des sous-programmes depuis tout language permettant d'appeler des procédures PL1 (FORTRAN, PL1, etc).

Liste et mode d'emploi des points d'entrée :

Point d'entrée : tabdon$ajouter

Ajouter un tableau dans une table de données.

On demande a l'utilisateur le "pathname" relatif de la table (on la crée si elle n'existe pas dans -wd).

Utilisation :

```
declare tabdon$ajouter entry (ptr, bin fixed(35), bin
fixed(35));
declare p_m ptr;
declare m(250000) bin fixed(35) based(p_m);
declare (lm, format) bin fixed(35);
call tabdon$ajouter (p_m, lm, format);
```

où :

```
          -- p_m = pointeur sur m = tableau à ajouter;
          -- lm=nombre de mots ;
          -- format :  0 - entiers ; 1 - reels ; 2 - chaines
de bits.
```

Point_d'entrée : tabdon$effacer
Effacer un tableau dans une table

Utilisation :
```
declare effacer entry (char(*) unaligned, char(*) unaligned);
call effacer (nom, numero);
```

Utilisation en commande interactive :
```
effacer nom numero
```

exemple :
```
call effacer (son_nom, "38");
```
ou bien :
```
effacer son_nom 38
```

-- nom : pathname de la table ;
-- numero : index du tableau dans le catalogue de la table (chaîne de caractères)
-- message d'erreur si la table n'existe pas, ou si l'index est inoccuppé
-- si numero<1 ou numero>100, aucune action

Point_d'entrée : tabdon$comprimer
Comprimer la table de données dont le nom est donné en paramètre.

Utilisation :
```
declare tabdon$comprimer entry (char(*) unaligned); call
tabdon$comprimer (nom);
```

Utilisation en commande interactive :
```
comprimer son_nom
```
-- message d'erreur si la table n'existe pas

Point_d'entrée : tabdon$lister
Lister les valeurs contenues dans l'un des tableaux d'une table.

Utilisation :
```
dcl tabdon$lister (char(*) unaligned, char(*) unaligned);
```

```
call tabdon$lister (nom, numero);
```

```
Utilisation en commande interactive :
lister nom numero
```

```
          -- nom = pathname relatif de la table
          -- numero = index du tableau dans la table.
          (chaîne de caract'eres).
          -- message d'erreur si la table n'existe pas, ou si
la position est libre
```

Point_d'entrée : lister_catalogue(nom);
Lister les entrées occuppées du catalogue d'une table.

Utilisation :

```
declare tabdon$lister_catalogue entry (char(*) unaligned);
call tabdon$lister_catalogue (nom);
```

```
Utilisation en commande interactive :
lister_catalogue nom
```

```
          -- nom : nom du segment contenant la table;
          -- message d'erreur si la table n'existe pas dans
-wd
```

Point_d'entrée : tabdon$lire_tab
Lire un tableau et le recopier dans le tableau fourni en paramètre

Utilisation :

```
declare tabdon$lire_tab entry (ptr, bin fixed(35),bin
fixed(35),bin fixed(35));
call lire_tab (p_m, lm, format, numero);
```

```
          -- p_m->m : tableau à recopier;
          -- lm = nombre de mots a transférer (bin
fixed(35));
          -- format 0 : entiers ; 1 : reels ; 2 : chaines
de 36 bits (bin fixed(35))
          -- numero : index dans la table (bin fixed (35))
```

-- le pathname de la table est demandé a l'utilisateur;
-- messages d'erreur :
----- table inexistante
----- numéro libre dans la table
----- non-correspondance entre les paramètres fournis et ceux lus dans la table.

ANNEXE C : LOGICIELS D'ANALYSE DES RESULTATS

Les programmes analyse, analyse_it et histo analysent l'histoire d'une éxécution muppy :
Ces programmes sont appelés sans arguments : les paramètres sont demandés à l'utilisateur lors de l'éxécution (utilisation normalement interactive).

1) programme_analyse

On cherche a repérer les instants passés par chaque processeur dans des sections de programme (generalement des sections critiques), que l'utilisateur a balisées (en entrée et en sortie) par des messages caractéristiques.

On imprime :

- la duree de chaque section critique;
- la moyenne de ces durées;
- l'écart-type de ces durées;

Le programme analyse lit un fichier resultat d'éxécution de MUPI : pour chaque ligne-message, on repère si elle correspond a l'une des transitions définies par l'utilisateur. (ces transitions sont définies par la procédure "scruter" fournie par l'utilisateur).

On peut observer plusieurs sections critiques.

Séquence d'appel :

analyse

Procedure_à_fournir_par_l'utilisateur_:

```
scruter:proc (ligne, p, c, trans );
/* paramètre d'entrée : */
dcl ligne char(132) varying;  /*  ligne à analyser */
/* paramètres de sortie :*/
dcl p     bin fixed(35);      /* numéro de processeur concerné */
dcl c     bin fixed(35);      /* numéro de champ */
dcl trans bit(1) aligned;
                /*  -- 1 : transition OK->non-OK
                    --      (fin de section critique)
                    -- 0 : transition non-OK->OK
                    --      (debut de section critique)                  */
......
......
end scruter;
```

Liste_des_questions_posées_à_l'utilisateur_:

quel est le nombre de processeurs observes ?
quel est le nombre de champs ?
nom du fichier a analyser?

```
titre SVP (132c. max entre quotes) ?
temps de depart ?
temps de fin ?
c = 1 a nbc :
       commentaire du champ n0.   ...?
Veux-tu observer les processeurs de 1 a nbp ?
si oui :
       faut-il les lister dans l'ordre ?
si non :
       p = 1 a nbp :
          processeur p en quelle colonne ?
si non :
       p = 1 a nbp :
          quel processeur en colonne p ?
lister la repartition des processeurs sur le listing (oui ou
non) ?
cette repartition est-elle satisfaisante  (oui ou non) ?
faudra-t-il compter les sections critiques entamees et non
terminees ?
(oui ou non) ?
lister toutes les valeurs (oui ou non) ?
```

2) Programme analyse_it

On cherche a mesurer le temps passé par chaque processeur entre des apparitions successives du même message (durée des itérations)

On imprime :

- la durée de chaque itération;
- la moyenne de ces durées;
- l'ecart-type de ces durées;

Le programme analyse_it lit un fichier resultat d'execution de MUPI : pour chaque ligne-message, on repère si elle correspond a l'un des messages repères définis par l'utilisateur. (ces messages sont définis par la procédure "scrut_it" fournie par l'utilisateur).

On peut observer plusieurs messages.

Séquence d'appel :

```
       analyse_it
```

Procedure à fournir par l'utilisateur :

```
scrut_it:proc (ligne, p, c) ;
/* paramètre d'entrê : */
dcl ligne char(132) varying;    /* ligne à analyser */
/* paramètres de sortie : */
dcl p bin fixed(35);            /* n0. de processeur concerné */
dcl c bin fixed(35);            /* n0. de champ */
......
```

```
.....
end scrut_it;
```

Liste des questions posées à l'utilisateur :

```
quel est le nombre de processeurs observes ?
quel est le nombre de messages a observer ?
nom du fichier a analyser?
titre SVP (132c. max entre quotes) ?
temps de depart ?
temps de fin ?
POUR c = 1 a nbc :
       commentaire du champ n0. c ?
       Veux-tu observer les processeurs de 1 a nbp ?
       SI OUI :
          faut-il les lister dans l'ordre ?
          SI NON : (pas dans l'ordre)
             POUR p = 1 à nbp :
                processeur p en quelle colonne ?

       SI NON (pas tous les processeurs de 1 a nbp) :
          POUR p = 1 a nbp :
             quel processeur en colonne p_col ?

lister la repartition des processeurs sur le listing (oui ou
non) ?
cette repartition est-elle satisfaisante  (oui ou non) ?
faudra-t-il compter les iterations entamees et non terminees
?
(oui ou non ?)
lister toutes les valeurs (oui ou non) ?
```

3) Programme histo :

Tracer sur TEKTRONICS l'histoire d'une execution muppy.

On cherche à repérer les instants passes par chaque processeur dans des sections de programme (généralement des sections critiques), que l'utilisateur a balisées (en entrée et en sortie) par des messages caracteristiques. Pour chaque section critique, on tracera un rectangle en face des heures correspondantes sur l'axe des temps.

Le programme histo lit un fichier résultat d'éxécution de MUPI : pour chaque ligne-message, on repère si elle correspond à l'une des transitions definies par l'utilisateur. (ces transitions sont definies par la procedure "scruter" fournie par l'utilisateur). On peut observer plusieurs sections critiques, chacune correspondant a un champ de l'image.

NOTES :

1) Quand le dessin est terminé, et que vous l'avez assez vu, tapez ";" suivi de NEWLINE, pour nettoyer l'écran et sortir du programme.

2) Le programme histo fait appel a la bibliothèque de tracés développée par A.MARROCCO (Rapport MODULEF, Septembre 1979).

procedure_à_fournir_par_l'utilisateur :

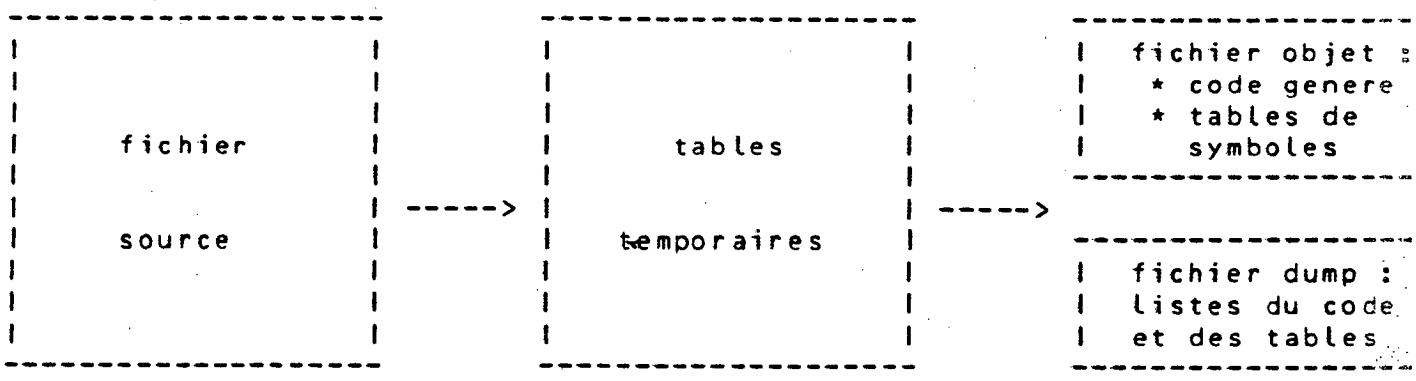
```
scruter (ligne, p, c, trans ) : proc;
/* paramètre d'entrée : */
dcl ligne char(132) varying;  /* ligne à analyser */
/* paramètres de sortie :    */
dcl p bin fixed(35);          /* n0. de processeur concerné */
dcl c bin fixed(35);          /* n0. de champ  */
dcl trans bit(1) aligned;     /* 1 : transition 0K->non-0K
                                    (fin de section critique)
                                 0 : transition non-0K->0K
                                    (debut de section critique) */

......
calcul de p, c, trans pour ligne
......
end scruter;
```

Liste_des_questions_posées_à_l'utilisateur :

```
quel est le nombre de processeurs observes ?
quel est le nombre de champs ?
nom du fichier à analyser?
titre SVP (70c. max entre quotes) ?
temps de depart ?
POUR c = 1 à nbc;
       commentaire du champ n0. ?
Voulez-vous observer les processeurs de 1 à nbp (oui ou non)
?
SI OUI :
          faut-il les dessiner dans l'ordre ?
          SI NON :
             POUR p = 1 à nbp;
                processeur p en quelle colonne ?
       SI NON (PAS TOUS LES PROCESSEURS ) :
          POUR p = 1 à nbp ;
             quel processeur en colonne p ?
lister la répartition des processeurs sur le dessin (oui ou
non) ?
cette répartition est-elle satisfaisante  (oui ou non) ?
```

ANNEXE D : ASSEMBLEUR : MODE D'EMPLOI



< fichier source > :
programme en assembleur MUPI

< tables temporaires > :
listes créées dynamiquement :
* des instructions;
* des variables locales;
* des paramètres;
* des étiquettes;
* des sous-programmes appelés;
* des constantes numériques;
* des constantes texte.

< fichier objet >
contient les structures suivantes : (tables définitives)

< programme >
tableau des instructions éxécutables
(une instruction par mot de 36 bits)

< marque_prog >
tableau des marques des instructions (pour PAUSE)

< proto_tb_symb>
prototype de table des symboles :
tableaux des symboles :
* tableau des paramètres;
* tableau des variables locales;
* tableau des étiquettes;
* tableau des constantes numériques;
* tableau des constantes textes;
* tableau des sous-programmes appelés.

Séquence d'appel :

```
asmupi <nom-du-programme> <options>
```

Le texte du source est dans le fichier :
<nom-du-programme>.source.
<options> ::= {map sc} | {nomap sc} | {map nosc} | {nomap nosc}
| {map} | {nomap} | {}

Signification des options :

map : production d'un fichier "dump" (listage du code généré et des tables de symboles);

nomap : pas de listage du code généré ni des symboles; (option par défaut)

sc : liste en écho au terminal des lignes lues sur le fichier source;

nosc : pas de liste en écho au terminal. (option par défaut)

Ainsi :
"asmupi nom"
est équivalent à :
"asmupi nom nomap nosc".

Remarque: Avec l'option map, toutes les lignes lues sont reproduites sur le fichier dump.

Le code généré et les tables de symboles sont stockés dans le segment : <nom-du-programme>.objet.

En cas d'erreur (de syntaxe, symbole non défini,...) un message est imprimé, et l'utilisateur est interrogé: il peut alors abandonner ou passer à l'instruction suivante; de toute façon, le fichier objet n'est pas créé.

Exemples de fichier dump :

LISTE DU TEXTE SOURCE :

```
   1 test:PROG
   2 i    : DEF IN(1),VE(BM1)   ! variable entiere sur le banc 1
   3 itab: DEF IN(10),HZ        ! 10 entiers stockes horizontalement
   4 a    : DEF RE(1),VE(BM2)   ! variable reelle sur le banc 2
   5 s    : DEF SM(1),VE(PR)    ! variable semaphore
   6 c    : CST IN,(1)          ! constante entiere
   7 c0   : CST IN,INIT         ! constante entiere initialisee au chargement
   8 t    : TEXTE (zxyvcwh)     ! un message significatif
   9
  10 STAT   D                   ! debut des prises de statistiques
  11 ILD    R0,c
  12 IST    R0,i
  13 MESS   t,i
  14 MESS   t,R0
  15 ILD    R1,c0
  16 IADD   R1,R0
  17 IST    R1,itab(1)
  18 MESS   t,itab(0)
  19 MESS   t,itab(1)
  20 RETURN
  21 END
```

```
LISTAGE DU CODE GENERE

                nom du sous programme :
                    test

                working-directory de l'utilitisateur:
                  >udd>Menusin>Thomasset>simulateur>gloss

                04/20/82  1647.3 hfe Tue


nombre d'instructions :          11
liste des instructions generees :
( * indique une instruction marquee (PAUSE) )
loc    code n     n     deplacement type          <>                    source
        op reg  symb.                ad  c. n.i.<>
   1   104  0        0          0   10   6   8  <>  STAT   D                    ! debut des prises de statistiques
   2     4  0       -1          0   11   0   4  <>  ILD    R0,c
   3     8  0        1          0   00   0   8  <>  IST    R0,i
   4    96  1        1          0   00   6   0  <>  MESS   t,i
   5    96  1        0          0   10   6   0  <>  MESS   t,R0
   6     4  1       -2          0   11   0   4  <>  ILD    R1,c0
   7     0  1        0          0   10   0   0  <>  IADD   R1,R0
   8     8  1        2          1   00   0   8  <>  IST    R1,itab(1)
   9    96  1        2          0   00   6   0  <>  MESS   t,itab(0)
  10    96  1        2          1   00   6   0  <>  MESS   t,itab(1)
  11    69  0        0          0   11   4  .5  <>  RETURN


nombre de parametres :                                    0
nombre de variables locales :                             4                                                    4
table des parametres & variables locales :
          VAR nom    type   extent   type de      init  <>       source
                                     stockage          <>
     1    0   i      00          1   0001   1  0 0    0<>i    : DEF IN(1),VE(BM1)  ! variable entiere sur le banc 1
     2    0   itab   00         10   0000   0  0 0    0<>itab: DEF IN(10),HZ      ! 10 entiers stockes horizontalement
     3    0   a      01          1   0001   2  0 0    0<>a    : DEF RE(1),VE(BM2)  ! variable reelle sur le banc 2
     4    0   s      10          1   0010   0  0 0    0<>s    : DEF SM(1),VE(PR)   ! variable semaphore


nombre de constantes numeriques :
                                  2
table des constantes numeriques :
          nom       valeur
       1  c                 1
       2  c0     : initialise a l'edition de liens


nombre de constantes :          1
table des constantes-texte :
          nom       valeur
       1  t         zxyvcwh
```

LISTE DU TEXTE SOURCE :

```
   1 test1:PROG
   2 i    : DEF IN(1),VE(BM1)   ! variable entiere sur le banc 1
   3 itab: DEF IN(10),HZ        ! 10 entiers stockes horizontalement
   4 c    : CST IN,INIT         ! constante entiere initialisee au chargement
   5 t    : TEXTE (abcdef)
   6
   7 STAT   D                   ! debut des prises de statistiques
   8 ETAT   8
   9 ILD    R0,c
  10 IST    R0,i
  11 IST    R0,itab(1)
  12 MESS   t,itab(1)
  13 RETURN
  14 END
```

```
LISTAGE DU CODE GENERE

                    nom du sous programme :
                        test1

                    working-directory de l'utilitisateur:
                      >udd>Menusin>Thomasset>simulateur>gloss

                    04/20/82  1647.6 hfe Tue


nombre d'instructions :          7
liste des instructions generees :
( * indique une instruction marquee (PAUSE) )
loc   code n     n     deplacement type        <>                    source
       op reg  symb.                ad  c. n.i.<>
   1  104  0        0          0   10   6   8  <>  STAT   D                  ! debut des prises de statistiques
   2  100  8        0          0   11   6   4  <>  ETAT   8
   3    4  0       -1          0   11   0   4  <>  ILD    R0,c
   4    8  0        1          0   00   0   8  <>  IST    R0,i
   5    8  0        2          1   00   0   8  <>  IST    R0,itab(1)
   6   96  1        2          1   00   6   0  <>  MESS   t,itab(1)
   7   69  0        0          0   11   4   5  <>  RETURN

119
nombre de parametres :                                    0
nombre de variables locales :                             2
table des parametres & variables locales :
          VAR nom    type   extent   type de      init  <>       source
                                     stockage          <>
     1    0   i      00          1   0001   1  0 0    0<>i    : DEF IN(1),VE(BM1)  ! variable entiere sur le banc 1
     2    0   itab   00         10   0000   0  0 0    0<>itab: DEF IN(10),HZ      ! 10 entiers stockes horizontalement


nombre de constantes numeriques :
                                  1
table des constantes numeriques :
          nom       valeur
       1  c       : initialise a l'edition de liens


nombre de constantes :        1
table des constantes-texte :
          nom       valeur
       1  t         abcdef


nombre d'etiquettes :       0


nombre de sous programmes appeles :         0
                    -.-.-.-.- FIN -.-.-.-.-
```

ANNEXE E : EXEMPLES DE SIMULATION MUPI

Séquence d'appel :

```
mupi    <nom>    <options>
<nom> :  nom d'un fichier objet (programme principal)
<options> : liste des options de simulation
            (pas de blanc entre les options;
            tout séparateur non blanc convient)
```

Liste des options :

Chaque option permet de modifier un ou plusieurs paramètres de simulation, dont on doit alors donner la valeur. Sinon, les paramètres utilisés sont ceux par défaut.

option	paramètre(s) modifiés
nb	nombres de processeurs et de bancs-memoire
tps	temps (des instructions, du réseau, de la mémoire)
zone	tailles de la zone d'allocation et de la mémoire du simulateur
mem	tailles des bancs-mémoire et niveau d'empilement
reel	execution des calculs flottants (ignorés sinon)
list(ls)	liste des paramètres de simulation utilisés.

Valeurs par défaut des paramètres de simulation :

```
nombre de processeurs.............................= 2
nombre de bancs-memoire...........................= 2

temps d'éxécution des instructions..............:
   arithmétique entiere...........................= 1
   arithmétique flottante.........................= 1
   logiques et décalages..........................= 1
   branchements...................................= 1
   privilégiées...................................= 1
   load et store..................................= 1
   call et return.................................= 1

temps du réseau.....................................:
   traversée......................................= 2
   mise en file d'attente.........................= 1
   sortie de file d'attente.......................= 1
```

```
temps de la mémoire (lecture/écriture).........= 2

taille des bancs-mémoire.......................= 100000
niveau d'empilement des sous-programmes........= 20
taille de la zone d'allocation du simulateur...= 10000
taille de la mémoire du simulateur.............= 2**10

calculs flottants non exécutés
```

Les exemples qui suivent correspondent aux sources donnés a l'annexe D.

```
mupi test nb/list


        SIMULATION DE MUPI
        nbp=**  nbm=**  ?? nbp=3 nbm=3;
        nbp=3 nbm=3;



        LISTE DES PARAMETRES UTILISES :
        -------------------------------


        nombre de processeurs ..............................:        3
        nombre de bancs memoire ............................:        3


        temps d'execution des instructions .................:
           instructions arithmetiques entieres .............:        1
           instructions arithmetiques flottantes ...........:        1
           instructions logiques & decalages ...............:        1
           branchements ....................................:        1
           instructions privilegiees .......................:        1
           load et store ...................................:        1
           call et return ..................................:        1


        temps du reseau ....................................:
           temps de traversee du reseau ....................:        2
           temps de mise en file d'attente .................:        1
           temps de sortie de file d'attente ...............:        1


        temps de reponse de la memoire (lecture ecriture) ..:        2


        taille des bancs memoire ...........................:   100000
        niveau max. d'empilement des sous-programmes .......:       20
        taille de la memoire du simulateur .................:  2**  10
        taille de la zone d'allocation du simulateur .......:    10000


        calculs flottants non executes


        ------------------------------------------------------------


        fichier objet utilise :
                >udd>Menusin>Thomasset>simulateur>gloss>test.objet
        initialisation de constantes numeriques dans le programme   test
        valeur de la constante          c0          de type   entier    ?
        10
        10
```

```
MESS: H=          17    P 1  PROG:test    TEXTE:zxyvcwh    i      (    0) =          1
MESS: H=          17    P 1  PROG:test    TEXTE:zxyvcwh    R 0 =
MESS:               1      7.450581e-009   00000000000000000000000000000001
MESS: H=          20    P 2  PROG:test    TEXTE:zxyvcwh    i      (    0) =          1
MESS: H=          20    P 2  PROG:test    TEXTE:zxyvcwh    R 0 =
MESS:               1      7.450581e-009   00000000000000000000000000000001
MESS: H=          23    P 3  PROG:test    TEXTE:zxyvcwh    i      (    0) =          1
MESS: H=          23    P 3  PROG:test    TEXTE:zxyvcwh    R 0 =
MESS:               1      7.450581e-009   00000000000000000000000000000001
MESS: H=          32    P 1  PROG:test    TEXTE:zxyvcwh    itab   (    0) =          0
MESS: H=          35    P 2  PROG:test    TEXTE:zxyvcwh    itab   (    0) =          0
MESS: H=          38    P 1  PROG:test    TEXTE:zxyvcwh    itab   (    1) =         11
MESS: H=          38    P 3  PROG:test    TEXTE:zxyvcwh    itab   (    0) =          0
MESS: H=          41    P 2  PROG:test    TEXTE:zxyvcwh    itab   (    1) =         11
MESS: H=          44    P 3  PROG:test    TEXTE:zxyvcwh    itab   (    1) =         11
STAT: nbp= 3   nbm= 3   de t1=          0   a t2=         45   duree=         45
SP 1: RQT:t=        33 %= 73 n=        5 BLO:t=         0 %=  0 n=        0 PRO:t=         6 %= 13 n=        0
SP 1: A 1:t=         4 %=  9 n=        1 A 2:t=         0 %=  0 n=        0
SP 1: A 3:t=         0 %=  0 n=        0
SP 2: RQT:t=        36 %= 80 n=        5 BLO:t=         0 %=  0 n=        0 PRO:t=         6 %= 13 n=        0
SP 2: A 1:t=         9 %= 20 n=        3 A 2:t=         2 %=  4 n=        2
SP 2: A 3:t=         0 %=  0 n=        0
SP 3: RQT:t=        39 %= 87 n=        5 BLO:t=         0 %=  0 n=        0 PRO:t=         6 %= 13 n=        0
SP 3: A 1:t=        12 %= 27 n=        3 A 2:t=         2 %=  4 n=        2
SP 3: A 3:t=         0 %=  0 n=        0
SB 1: LIB:t=        16 %= 36 n=        0 OSC:t=        11 %= 24 n=        2 OAC:t=        18 %= 40 n=        7
SB 1: C 2:t=        11 %= 24 n=        3 C 3:t=         7 %= 16 n=        4
SB 2: LIB:t=        25 %= 56 n=        0 OSC:t=        16 %= 36 n=        2 OAC:t=         4 %=  9 n=        4
SB 2: C 2:t=         4 %=  9 n=        4 C 3:t=         0 %=  0 n=        0
SB 3: LIB:t=        45 %=100 n=        0 OSC:t=         0 %=  0 n=        0 OAC:t=         0 %=  0 n=        0
SB 3: C 2:t=         0 %=  0 n=        0 C 3:t=         0 %=  0 n=        0

FIN DE LA SIMULATION DE MUPI
```

```
mupi test1 nb/list

    SIMULATION DE MUPI
    nbp=**  nbm=**  ?? nbp=3 nbm=3;
    nbp=3 nbm=3;



   LISTE DES PARAMETRES UTILISES :
   -------------------------------


   nombre de processeurs ..........................:          3
   nombre de bancs memoire ........................:          3


   temps d'execution des instructions .............:
      instructions arithmetiques entieres .........:          1
      instructions arithmetiques flottantes .......:          1
      instructions logiques & decalages ...........:          1
      branchements ................................:          1
      instructions privilegiees ...................:          1
      load et store ...............................:          1
      call et return ..............................:          1


   temps du reseau ................................:
      temps de traversee du reseau ................:          2
      temps de mise en file d'attente .............:          1
      temps de sortie de file d'attente ...........:          1


   temps de reponse de la memoire (lecture ecriture) .:       2


   taille des bancs memoire .......................:     100000
   niveau max. d'empilement des sous-programmes ...:         20
   taille de la memoire du simulateur .............:    2**  10
   taille de la zone d'allocation du simulateur ...:      10000


   calculs flottants non executes


   ------------------------------------------------------------


   fichier objet utilise :
            >udd>Menusin>Thomasset>simulateur>gloss>test1.objet
   initialisation de constantes numeriques dans le programme   test1
   valeur de la constante          c          de type   entier    ?
   10
   10
```

```
ETAT: H=    0   niveau= 8  DT=       0  DK=       0
EVT:  H     0   PP    P 2   B 0
RES:  H=    0   P: L   L   L     B: L L L
P 2:  H=    0   LO   CC= 0   CO=test1 I   2 II STAT  2 I R 0 N 0 D   0 T2
ETAT: H=    0   niveau= 8  DT=       0  DK=       0
EVT:  H     0   PP    P 3   B 0
RES:  H=    0   P: L   L   L     B: L L L
P 3:  H=    0   LO   CC= 0   CO=test1 I   2 II STAT  2 I R 0 N 0 D   0 T2
ETAT: H=    0   niveau= 8  DT=       0  DK=       0
EVT:  H     0   PP    P 1   B 0
RES:  H=    0   P: L   L   L     B: L L L
P 1:  H=    0   LO   CC= 0   CO=test1 I   3 II ETAT  2 I R 8 N 0 D   0 T3
EVT:  H     0   PP    P 2   B 0
RES:  H=    0   P: L   L   L     B: L L L
P 2:  H=    0   LO   CC= 0   CO=test1 I   3 II ETAT  2 I R 8 N 0 D   0 T3
EVT:  H     0   PP    P 3   B 0
RES:  H=    0   P: L   L   L     B: L L L
P 3:  H=    0   LO   CC= 0   CO=test1 I   3 II ETAT  2 I R 8 N 0 D   0 T3
EVT:  H     1   PP    P 1   B 0
RES:  H=    1   P: L   L   L     B: L L L
P 1:  H=    1   LO   CC= 0   CO=test1 I   4 II ILD   2 I R 0 N-1 D   0 T3
EVT:  H     1   PP    P 2   B 0
RES:  H=    1   P: L   L   L     B: L L L
P 2:  H=    1   LO   CC= 0   CO=test1 I   4 II ILD   2 I R 0 N-1 D   0 T3
EVT:  H     1   PP    P 3   B 0
RES:  H=    1   P: L   L   L     B: L L L
P 3:  H=    1   LO   CC= 0   CO=test1 I   4 II ILD   2 I R 0 N-1 D   0 T3
EVT:  H     2   RRO   P 1   B 1   E   S=        2
RES:  H=    2   P: L   L   L     B: L L L
P 1:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     2   RRO   P 2   B 1   E   S=        2
RES:  H=    2   P: 0 1 L   L     B: 0 L L
P 2:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     2   RRO   P 3   B 1   E   S=        2
RES:  H=    2   P: 0 1 A 1 L     B: 0 L L
P 3:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     4   BRO   P 1   B 1   E   S=        2
RES:  H=    4   P: 0 1 A 1 A 1   B: 0 L L
P 1:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     6   RAO   P 1   B 1   E   S=        2
RES:  H=    6   P: 0 1 A 1 A 1   B: 0 L L
P 1:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     7   BRO   P 2   B 1   E   S=        2
RES:  H=    7   P: L   0 1 A 1   B: 0 L L
P 2:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     8   PAO   P 1   B 1   E   S=        2
RES:  H=    8   P: L   0 1 A 1   B: 0 L L
P 1:  H=    2   AO   CC= 0   CO=test1 I   4 II IST   8 I R 0 N 1 D   0 T0
EVT:  H     8   PP    P 1   B 0
RES:  H=    8   P: L   0 1 A 1   B: 0 L L
```

```
P 1:  H=     8   LO   CC= 0   C0=test1 I   5 II IST   2 I R 0 N 1 D   0 TO
EVT:  H      9   RAO   P 2   B 1   E   S=        2
RES:  H=     9   P: L   0 1 A 1   B: 0 L L
P 2:  H=     2   AO   CC= 0   C0=test1 I   4 II IST   8 I R 0 N 1 D   0 TO
EVT:  H      9   RRO   P 1   B 2   E   S=        4
RES:  H=     9   P: L   L   0 1   B: 0 L L
P 1:  H=     9   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     10   BRO   P 3   B 1   E   S=        2
RES:  H=    10   P: 0 2 L   0 1   B: 0 0 L
P 3:  H=     2   AO   CC= 0   C0=test1 I   4 II IST   8 I R 0 N 1 D   0 TO
EVT:  H     11   PAO   P 2   B *   E   S=        2
RES:  H=    11   P: 0 2 L   0 1   B: 0 0 L
P 2:  H=     2   AO   CC= 0   C0=test1 I   4 II IST   8 I R 0 N 1 D   0 TO
EVT:  H     11   BRO   P 1   B 2   E   S=        4
RES:  H=    11   P: 0 2 L   0 1   B: 0 0 L
P 1:  H=     9   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     11   PP   P 2   B 0
RES:  H=    11   P: 0 2 L   0 1   B: 0 0 L
P 2:  H=    11   LO   CC= 0   C0=test1 I   5 II IST   2 I R 0 N 1 D   0 TO
EVT:  H     12   RAO   P 3   B 1   E   S=        2
RES:  H=    12   P: 0 2 L   0 1   B: 0 0 L
P 3:  H=     2   AO   CC= 0   C0=test1 I   4 II IST   8 I R 0 N 1 D   0 TO
EVT:  H     12   RRO   P 2   B 2   E   S=        4
RES:  H=    12   P: 0 2 L   L   B: L 0 L
P 2:  H=    12   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     13   RAO   P 1   B 2   E   S=        4
RES:  H=    13   P: 0 2 A 2 L   B: L 0 L
P 1:  H=     9   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     14   PAO   P 3   B 1   E   S=        2
RES:  H=    14   P: L   0 2 L   B: L 0 L
P 3:  H=     2   AO   CC= 0   C0=test1 I   4 II IST   8 I R 0 N 1 D   0 TO
EVT:  H     14   BRO   P 2   B 2   E   S=        4
RES:  H=    14   P: L   0 2 L   B: L 0 L
P 2:  H=    12   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     14   PP   P 3   B 0
RES:  H=    14   P: L   0 2 L   B: L 0 L
P 3:  H=    14   LO   CC= 0   C0=test1 I   5 II IST   2 I R 0 N 1 D   0 TO
EVT:  H     15   PAO   P 1   B 2   E   S=        4
RES:  H=    15   P: L   0 2 L   B: L 0 L
P 1:  H=     9   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     15   RRO   P 3   B 2   E   S=        4
RES:  H=    15   P: L   0 2 L   B: L 0 L
P 3:  H=    15   AO   CC= 0   C0=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H     15   PP   P 1   B 0
RES:  H=    15   P: L   0 2 A 2   B: L 0 L
P 1:  H=    15   LO   CC= 0   C0=test1 I   6 II IST   2 I R 0 N 2 D   1 TO
EVT:  H     15   RRO   P 1   B 2   L   S=        4
RES:  H=    15   P: L   0 2 A 2   B: L 0 L
P 1:  H=    15   AO   CC= 0   C0=test1 I   6 II MESS  6 I R 1 N 2 D   1 TO
EVT:  H     16   RAO   P 2   B 2   E   S=        4
RES:  H=    16   P: A 2 0 2 A 2   B: L 0 L
```

```
P 2:  H=      12   AO   CC= 0   CO=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H       17   BRO   P 3   B 2   E   S=        4
RES:  H=      17   P: A 2 L   0 2   B: L 0 L
P 3:  H=      15   AO   CC= 0   CO=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H       18   PAO   P 2   B 2   E   S=        4
RES:  H=      18   P: A 2 L   0 2   B: L 0 L
P 2:  H=      12   AO   CC= 0   CO=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H       18   PP   P 2   B 0
RES:  H=      18   P: A 2 L   0 2   B: L 0 L
P 2:  H=      18   LO   CC= 0   CO=test1 I   6 II IST   2 I R 0 N 2 D   1 TO
EVT:  H       18   RRO   P 2   B 2   L   S=        4
RES:  H=      18   P: A 2 L   0 2   B: L 0 L
P 2:  H=      18   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       19   RAO   P 3   B 2   E   S=        4
RES:  H=      19   P: A 2 A 2 0 2   B: L 0 L
P 3:  H=      15   AO   CC= 0   CO=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H       20   BRO   P 1   B 2   L   S=        4
RES:  H=      20   P: 0 2 A 2 L   B: L 0 L
P 1:  H=      15   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       21   PAO   P 3   B 2   E   S=        4
RES:  H=      21   P: 0 2 A 2 L   B: L 0 L
P 3:  H=      15   AO   CC= 0   CO=test1 I   5 II IST   8 I R 0 N 2 D   1 TO
EVT:  H       21   PP   P 3   B 0
RES:  H=      21   P: 0 2 A 2 L   B: L 0 L
P 3:  H=      21   LO   CC= 0   CO=test1 I   6 II IST   2 I R 0 N 2 D   1 TO
EVT:  H       21   RRO   P 3   B 2   L   S=        4
RES:  H=      21   P: 0 2 A 2 L   B: L 0 L
P 3:  H=      21   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       22   RAO   P 1   B 2   L   S=        4
RES:  H=      22   P: 0 2 A 2 A 2   B: L 0 L
P 1:  H=      15   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       23   BRO   P 2   B 2   L   S=        4
RES:  H=      23   P: L   0 2 A 2   B: L 0 L
P 2:  H=      18   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       24   PAO   P 1   B 2   L   S=        4
RES:  H=      24   P: L   0 2 A 2   B: L 0 L
P 1:  H=      15   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
MESS: H=      24   P 1  PROG:test1  TEXTE:abcdef   itab (  1) =      10
EVT:  H       24   PP   P 1   B 0
RES:  H=      24   P: L   0 2 A 2   B: L 0 L
P 1:  H=      24   LO   CC= 0   CO=test1 I   7 II MESS   2 I R 1 N 2 D   1 TO
EVT:  H       25   RAO   P 2   B 2   L   S=        4
RES:  H=      25   P: L   0 2 A 2   B: L 0 L
P 2:  H=      18   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       26   BRO   P 3   B 2   L   S=        4
RES:  H=      26   P: L   L   0 2   B: L 0 L
P 3:  H=      21   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
EVT:  H       27   PAO   P 2   B 2   L   S=        4
RES:  H=      27   P: L   L   0 2   B: L 0 L
P 2:  H=      18   AO   CC= 0   CO=test1 I   6 II MESS   6 I R 1 N 2 D   1 TO
MESS: H=      27   P 2  PROG:test1  TEXTE:abcdef   itab (  1) =      10
```

```
EVT:  H          27   PP     P 2    B 0
RES:  H=         27   P: L    L     0 2    B: L 0 L
P 2:  H=         27   L0    CC= 0    C0=test1 I    7 II MESS   2 I R 1 N 2 D    1 TO
EVT:  H          28   RA0    P 3    B 2    L    S=          4
RES:  H=         28   P: L    L     0 2    B: L 0 L
P 3:  H=         21   A0    CC= 0    C0=test1 I    6 II MESS   6 I R 1 N 2 D    1 TO
EVT:  H          30   PA0    P 3    B 2    L    S=          4
RES:  H=         30   P: L    L     L      B: L L L
P 3:  H=         21   A0    CC= 0    C0=test1 I    6 II MESS   6 I R 1 N 2 D    1 TO
MESS: H=         30   P 3   PROG:test1   TEXTE:abcdef     itab  (   1) =        10
EVT:  H          30   PP     P 3    B 0
RES:  H=         30   P: L    L     L      B: L L L
P 3:  H=         30   L0    CC= 0    C0=test1 I    7 II MESS   2 I R 1 N 2 D    1 TO
STAT: nbp= 3  nbm= 3   de t1=          0   a t2=        31   duree=        31
SP 1: RQT:t=       21 %= 68 n=       3 BL0:t=         0 %=  0 n=       0 PRO:t=         4 %= 13 n=       0
SP 1: A 1:t=        0 %=  0 n=       0 A 2:t=         4 %= 13 n=       1
SP 1: A 3:t=        0 %=  0 n=       0
SP 2: RQT:t=       24 %= 77 n=       3 BL0:t=         0 %=  0 n=       0 PRO:t=         4 %= 13 n=       0
SP 2: A 1:t=        4 %= 13 n=       1 A 2:t=         5 %= 16 n=       2
SP 2: A 3:t=        0 %=  0 n=       0
SP 3: RQT:t=       27 %= 87 n=       3 BL0:t=         0 %=  0 n=       0 PRO:t=         4 %= 13 n=       0
SP 3: A 1:t=        7 %= 23 n=       1 A 2:t=         5 %= 16 n=       2
SP 3: A 3:t=        0 %=  0 n=       0
SB 1: LIB:t=       21 %= 68 n=       0 OSC:t=         3 %= 10 n=       1 OAC:t=         7 %= 23 n=       2
SB 1: C 2:t=        3 %= 10 n=       1 C 3:t=         4 %= 13 n=       1
SB 2: LIB:t=       12 %= 39 n=       0 OSC:t=         8 %= 26 n=       1 OAC:t=        11 %= 35 n=       5
SB 2: C 2:t=        8 %= 26 n=       2 C 3:t=         3 %= 10 n=       3
SB 3: LIB:t=       31 %=100 n=       0 OSC:t=         0 %=  0 n=       0 OAC:t=         0 %=  0 n=       0
SB 3: C 2:t=        0 %=  0 n=       0 C 3:t=         0 %=  0 n=       0

FIN DE LA SIMULATION DE MUPI
```

BIBLIOGRAPHIE

(1) J. ERHEL (1982)
"Parallelisation d'algorithmes numeriques", These de 3eme cycle, Universite Paris XI, 11 Mars 1982.

(2) M. FLYNN (1972).
"Some computer organizations and their effectiveness", IEEE Trans. on Computers, C-21, 9 (Sept. 1972), pp. 948-960.

(3) WULF and BELL (1972).
"C.mmp -- A Multi-Mini-Processor", Proceedings AFIPS 1972 FJCC, Vol. 41, AFIPS Press, pp. 765-777.

(4) WULF, COHEN, CORWIN, JONES, LEVIN, PIERSON, POLLACK (1974).
"Hydra : the Kernel of a Multiprocessor Operating System" Communications of the ACM, 17, 6, 1974, pp. 337-345.

==========

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique